Boost.Math

Table of Contents

Overview

The Greatest Common Divisor and Least Common Multiple library provides run-time and compile-time evaluation of the greatest common divisor (GCD) or least common multiple (LCM) of two integers.

The Special Functions library currently provides eight templated special functions, in namespace boost. Two of these (sinc_pi and sinhc_pi) are needed by our implementation of quaternions and octonions. The functions acosh, asinh and atanh are entirely classical, the function sinc_pi sees heavy use in signal processing tasks, and the function sinhc_pi is an ad'hoc function whose naming is modelled on sinc_pi and hyperbolic functions. The functions log1p, expm1 and hypot are all part of the C99 standard but not yet C++. Two of these (log1p and hypot) were needed for the complex number inverse trigonometric functions.

The Complex Number Inverse Trigonometric Functions are the inverses of trigonometric functions currently present in the C++ standard. Equivalents to these functions are part of the C99 standard, and they will be part of the forthcoming Technical Report on C++ Standard Library Extensions.

Quaternions are a relative of complex numbers often used to parameterise rotations in three dimentional space.

Octonions, like quaternions, are a relative of complex numbers. Octonions see some use in theoretical physics.

Greatest Common Divisor and Least Common Multiple

Introduction

The class and function templates in <boost/math/common_factor.hpp> provide run-time and compile-time evaluation of the greatest common divisor (GCD) or least common multiple (LCM) of two integers. These facilities are useful for many numeric-oriented generic programming problems.

Synopsis

```
namespace boost
{
namespace math
{

template < typename IntegerType >
   class gcd_evaluator;
template < typename IntegerType >
   class lcm_evaluator;

template < typename IntegerType >
   IntegerType  gcd( IntegerType const &a, IntegerType const &b );
template < typename IntegerType >
   IntegerType  lcm( IntegerType const &a, IntegerType const &b );

template < unsigned long Value1, unsigned long Value2 >
   struct static_gcd;
template < unsigned long Value1, unsigned long Value2 >
   struct static_lcm;

}
}
```

GCD Function Object

Header: <boost/math/common_factor_rt.hpp>

```
template < typename IntegerType >
class boost::math::gcd_evaluator
{
public:
   // Types
   typedef IntegerType  result_type;
   typedef IntegerType  first_argument_type;
   typedef IntegerType  second_argument_type;

   // Function object interface
   result_type  operator ()( first_argument_type const &a,
   second_argument_type const &b ) const;
};
```

The boost::math::gcd_evaluator class template defines a function object class to return the greatest common divisor of two integers. The template is parameterized by a single type, called IntegerType here. This type should be a numeric type that represents integers. The result of the function object is always nonnegative, even if either of the operator arguments is negative.

This function object class template is used in the corresponding version of the GCD function template. If a numeric type wants to customize evaluations of its greatest common divisors, then the type should specialize on the gcd_evaluator class template.

LCM Function Object

Header: <boost/math/common_factor_rt.hpp>

```
template < typename IntegerType >
class boost::math::lcm_evaluator
{
public:
   // Types
   typedef IntegerType  result_type;
   typedef IntegerType  first_argument_type;
   typedef IntegerType  second_argument_type;

   // Function object interface
   result_type  operator ()( first_argument_type const &a,
   second_argument_type const &b ) const;
};
```

The boost::math::lcm_evaluator class template defines a function object class to return the least common multiple of two integers. The template is parameterized by a single type, called IntegerType here. This type should be a numeric type that represents integers. The result of the function object is always nonnegative, even if either of the operator arguments is negative. If the least common multiple is beyond the range of the integer type, the results are undefined.

This function object class template is used in the corresponding version of the LCM function template. If a numeric type wants to customize evaluations of its least common multiples, then the type should specialize on the lcm_evaluator class template.

Run-time GCD & LCM Determination

Header: <boost/math/common_factor_rt.hpp>

```
template < typename IntegerType >
IntegerType  boost::math::gcd( IntegerType const &a, IntegerType const &b );

template < typename IntegerType >
IntegerType  boost::math::lcm( IntegerType const &a, IntegerType const &b );
```

The boost::math::gcd function template returns the greatest common (nonnegative) divisor of the two integers passed to it. The boost::math::lcm function template returns the least common (nonnegative) multiple of the two integers passed to it. The function templates are parameterized on the function arguments' IntegerType, which is also the return type. Internally, these function templates use an object of the corresponding version of the gcd_evaluator and lcm_evaluator class templates, respectively.

Compile time GCD and LCM determination

Header: <boost/math/common_factor_ct.hpp>

```
template < unsigned long Value1, unsigned long Value2 >
struct boost::math::static_gcd
{
   static unsigned long const  value = implementation_defined;
};

template < unsigned long Value1, unsigned long Value2 >
struct boost::math::static_lcm
{
   static unsigned long const  value = implementation_defined;
};
```

The boost::math::static_gcd and boost::math::static_lcm class templates take two value-based template parameters of the unsigned long type and have a single static constant data member, value, of that same type. The value of that member is the greatest common

factor or least common multiple, respectively, of the template arguments. A compile-time error will occur if the least common multiple is beyond the range of an unsigned long.

Example

```
#include <boost/math/common_factor.hpp>
#include <algorithm>
#include <iterator>


int main()
{
   using std::cout;
   using std::endl;

   cout << "The GCD and LCM of 6 and 15 are "
   << boost::math::gcd(6, 15) << " and "
   << boost::math::lcm(6, 15) << ", respectively."
   << endl;

   cout << "The GCD and LCM of 8 and 9 are "
   << boost::math::static_gcd<8, 9>::value
   << " and "
   << boost::math::static_lcm<8, 9>::value
   << ", respectively." << endl;

   int  a[] = { 4, 5, 6 }, b[] = { 7, 8, 9 }, c[3];
   std::transform( a, a + 3, b, c, boost::math::gcd_evaluator<int>() );
   std::copy( c, c + 3, std::ostream_iterator<int>(cout, " ") );
}
```

Header <boost/math/common_factor.hpp>

This header simply includes the headers <boost/math/common_factor_ct.hpp> and <boost/math/common_factor_rt.hpp>.

Note this is a legacy header: it used to contain the actual implementation, but the compile-time and run-time facilities were moved to separate headers (since they were independent of each other).

Demonstration Program

The program common_factor_test.cpp is a demonstration of the results from instantiating various examples of the run-time GCD and LCM function templates and the compile-time GCD and LCM class templates. (The run-time GCD and LCM class templates are tested indirectly through the run-time function templates.)

Rationale

The greatest common divisor and least common multiple functions are greatly used in some numeric contexts, including some of the other Boost libraries. Centralizing these functions to one header improves code factoring and eases maintainence.

History

- 17 Dec 2005: Converted documentation to Quickbook Format.
- 2 Jul 2002: Compile-time and run-time items separated to new headers.
- 7 Nov 2001: Initial version

Credits

The author of the Boost compilation of GCD and LCM computations is Daryle Walker. The code was prompted by existing code hiding in the implementations of Paul Moore's rational library and Steve Cleary's pool library. The code had updates by Helmut Zeisel.

Math Special Functions

Overview

The Special Functions library currently provides eight templated special functions, in namespace boost. Two of these (sinc_pi and sinhc_pi) are needed by our implementation of quaternions and octonions.

The functions acosh, asinh and atanh are entirely classical, the function sinc_pi sees heavy use in signal processing tasks, and the function sinhc_pi is an ad'hoc function whose naming is modelled on sinc_pi and hyperbolic functions.

The functions log1p, expm1 and hypot are all part of the C99 standard but not yet C++. Two of these (log1p and hypot) were needed for the complex number inverse trigonometric functions.

The functions implemented here can throw standard exceptions, but no exception specification has been made.

Header Files

The interface and implementation for each function (or forms of a function) are both supplied by one header file:

- acosh.hpp
- asinh.hpp
- atanh.hpp
- expm1.hpp
- hypot.hpp
- log1p.hpp
- sinc.hpp
- sinhc.hpp

Synopsis

```
namespace boost{ namespace math{

template<typename T>
T acosh(const T x);

template<typename T>
T asinh(const T x);

template<typename T>
T atanh(const T x);

template<typename T>
T expm1(const T x);

template<typename T>
T hypot(const T x);

template<typename T>
T log1p(const T x);

template<typename T>
T sinc_pi(const T x);

template<typename T, template<typename> class U>
U<T> sinc_pi(const U<T> x);

template<typename T>
T sinhc_pi(const T x);

template<typename T, template<typename> class U>
U<T> sinhc_pi(const U<T> x);

}
}
```

acosh

```
template<typename T>
T acosh(const T x);
```

Computes the reciprocal of (the restriction to the range of `[0;+ [`) the hyperbolic cosine function, at x. Values returned are positive. Generalised Taylor series are used near 1 and Laurent series are used near the infinity to ensure accuracy.

If x is in the range `]- ;+1[` a quiet NaN is returned (if the system allows, otherwise a `domain_error` exception is generated).

asinh

```
template<typename T>
T asinh(const T x);
```

Computes the reciprocal of the hyperbolic sine function. Taylor series are used at the origin and Laurent series are used near the infinity to ensure accuracy.

atanh

```
template<typename T>
T atanh(const T x);
```

Computes the reciprocal of the hyperbolic tangent function, at x. Taylor series are used at the origin to ensure accuracy.

If x is in the range `]- ;-1[` or in the range `]+1;+ [` a quiet NaN is returned (if the system allows, otherwise a `domain_error` exception is generated).

If x is in the range `[-1;-1+ [`, minus infinity is returned (if the system allows, otherwise an `out_of_range` exception is generated), with ε denoting numeric_limits<T>::epsilon().

If x is in the range `]+1- ;+1]`, plus infinity is returned (if the system allows, otherwise an `out_of_range` exception is generated), with ε denoting numeric_limits<T>::epsilon().

expm1

```
template <class T>
T expm1(T t);
```

Effects: returns $e^x - 1$.

For small x, then e^x is very close to 1, as a result calculating $e^x - 1$ results in catastrophic cancellation errors when x is small. `expm1` calculates $e^x - 1$ using a series expansion when x is small (giving an accuracy of less than 2ε).

Finally when BOOST_HAS_EXPM1 is defined then the `float/double/long double` specializations of this template simply forward to the platform's native implementation of this function.

hypot

```
template <class T>
T hypot(T x, T y);
```

Effects: computes $\sqrt{x^2 + y^2}$ in such a way as to avoid undue underflow and overflow.

When calculating $\sqrt{x^2 + y^2}$ it's quite easy for the intermediate terms to either overflow or underflow, even though the result is in fact perfectly representable. One possible alternative form is $\sqrt{x}\sqrt{y}\sqrt{\frac{x}{y} + \frac{y}{x}}$, but that can overflow or underflow if x and y are of very differing magnitudes. The `hypot` function takes care of all the special cases for you, so you don't have to.

log1p

```
template <class T>
T log1p(T x);
```

Effects: returns the natural logarithm of `x+1`.

There are many situations where it is desirable to compute `log(x+1)`. However, for small `x` then `x+1` suffers from catastrophic cancellation errors so that `x+1 == 1` and `log(x+1) == 0`, when in fact for very small x, the best approximation to `log(x+1)` would be `x`. `log1p` calculates the best approximation to `log(1+x)` using a Taylor series expansion for accuracy (less than 2ε). Note that there are faster methods available, for example using the equivalence:

```
log(1+x) == (log(1+x) * x) / ((1-x) - 1)
```

However, experience has shown that these methods tend to fail quite spectacularly once the compiler's optimizations are turned on. In contrast, the series expansion method seems to be reasonably immune optimizer-induced errors.

Finally when BOOST_HAS_LOG1P is defined then the `float/double/long double` specializations of this template simply forward to the platform's native implementation of this function.

sinc_pi

```
template<typename T>
T sinc_pi(const T x);

template<typename T, template<typename> class U>
U<T> sinc_pi(const U<T> x);
```

Computes the Sinus Cardinal of x. The second form is for complexes, quaternions, octonions... Taylor series are used at the origin to ensure accuracy.

sinhc_pi

```
template<typename T>
T sinhc_pi(const T x);

template<typename T, template<typename> class U>
U<T> sinhc_pi(const U<T> x);
```

Computes the Hyperbolic Sinus Cardinal of x. The second form is for complexes, quaternions, octonions... Taylor series are used at the origin to ensure accuracy.

Test Programs

The special_functions_test.cpp and log1p_expm1_test.cpp test programs test the functions for float, double and long double arguments (sample output, with message output enabled).

If you define the symbol BOOST_SPECIAL_FUNCTIONS_TEST_VERBOSE, you will get additional output (verbose output), which may prove useful for tuning on your platform (the library use "reasonable" tolerances, which may prove to be too strict for your platform); this will only be helpfull if you enable message output at the same time, of course (by uncommenting the relevant line in the test or by adding `--log_level=messages` to your command line,...).

Acknowledgements

The mathematical text has been typeset with Nisus Writer, and the illustrations have been made with Graphing Calculator. Jens Maurer was the Review Manager for this library. More acknowledgements in the History section. Thank you to all who contributed to the discution about this library.

History

- 1.5.0 - 17/12/2005: John Maddock added log1p, expm1 and hypot. Converted documentation to Quickbook Format.
- 1.4.2 - 03/02/2003: transitionned to the unit test framework; <boost/config.hpp> now included by the library headers (rather than the test files).
- 1.4.1 - 18/09/2002: improved compatibility with Microsoft compilers.
- 1.4.0 - 14/09/2001: added (after rewrite) acosh and asinh, which were submited by Eric Ford; applied changes for Gcc 2.9.x suggested by John Maddock; improved accuracy; sanity check for test file, related to accuracy.
- 1.3.2 - 07/07/2001: introduced namespace math.
- 1.3.1 - 07/06/2001:(end of Boost review) split special_functions.hpp into atanh.hpp, sinc.hpp and sinhc.hpp; improved efficiency of atanh with compile-time technique (Daryle Walker); improved accuracy of all functions near zero (Peter Schmitteckert).
- 1.3.0 - 26/03/2001: support for complexes & all, for cardinal functions.
- 1.2.0 - 31/01/2001: minor modifications for Koenig lookup.
- 1.1.0 - 23/01/2001: boostification.
- 1.0.0 - 10/08/1999: first public version.

To Do

- Add more functions.
- Improve test of each function.

Complex Number Inverse Trigonometric Functions

The following complex number algorithms are the inverses of trigonometric functions currently present in the C++ standard. Equivalents to these functions are part of the C99 standard, and will be part of the forthcoming Technical Report on C++ Standard Library Extensions.

Implementation and Accuracy

Although there are deceptively simple formulae available for all of these functions, a naive implementation that used these formulae would fail catastrophically for some input values. The Boost versions of these functions have been implemented using the methodology described in "Implementing the Complex Arcsine and Arccosine Functions Using Exception Handling" by T. E. Hull Thomas F. Fairgrieve and Ping Tak Peter Tang, ACM Transactions on Mathematical Software, Vol. 23, No. 3, September 1997. This means that the functions are well defined over the entire complex number range, and produce accurate values even at the extremes of that range, where as a naive formula would cause overflow or underflow to occur during the calculation, even though the result is actually a representable value. The maximum theoretical relative error for all of these functions is less than 9.5E for every machine-representable point in the complex plane. Please refer to comments in the header files themselves and to the above mentioned paper for more information on the implementation methodology.

asin

Header:

```
#include <boost/math/complex/asin.hpp>
```

Synopsis:

```
template<class T>
std::complex<T> asin(const std::complex<T>& z);
```

Effects: returns the inverse sine of the complex number z.

Formula: $\sin^{-1}(z) = -i\log(iz + \sqrt{1 - z^2})$

acos

Header:

```
#include <boost/math/complex/acos.hpp>
```

Synopsis:

```
template<class T>
std::complex<T> acos(const std::complex<T>& z);
```

Effects: returns the inverse cosine of the complex number z.

Formula: $\cos^{-1}(z) = \frac{\pi}{2} + i\log(iz + \sqrt{1 - z^2})$

atan

Header:

```
#include <boost/math/complex/atan.hpp>
```

Synopsis:

```
template<class T>
std::complex<T> atan(const std::complex<T>& z);
```

Effects: returns the inverse tangent of the complex number z.

Formula: $\tan^{-1}(z) = \frac{i}{2}(\log(1 - iz) - \log(iz + 1)) = -i\tanh^{-1}(iz)$

asinh

Header:

```
#include <boost/math/complex/asinh.hpp>
```

Synopsis:

```
template<class T>
std::complex<T> asinh(const std::complex<T>& z);
```

Effects: returns the inverse hyperbolic sine of the complex number z.

Formula: $\sinh^{-1}(z) = \log(z + \sqrt{z^2 + 1}) = i\sin^{-1}(-iz)$

acosh

Header:

```
#include <boost/math/complex/acosh.hpp>
```

Synopsis:

```
template<class T>
std::complex<T> acosh(const std::complex<T>& z);
```

Effects: returns the inverse hyperbolic cosine of the complex number z.

Formula: $\cosh^{-1}(z) = \log(z + \sqrt{z-1}\sqrt{z+1}) = \pm i\cos^{-1}(z)$

atanh

Header:

```
#include <boost/math/complex/atanh.hpp>
```

Synopsis:

```
template<class T>
std::complex<T> atanh(const std::complex<T>& z);
```

Effects: returns the inverse hyperbolic tangent of the complex number z.

Formula: $\tanh^{-1}(z) = \frac{1}{2}(\log(1+z) - \log(1-z))$

History

- 2005/12/17: Added support for platforms with no meaningful numeric_limits<>::infinity().
- 2005/12/01: Initial version, added as part of the TR1 library.

Quaternions

Overview

Quaternions are a relative of complex numbers.

Quaternions are in fact part of a small hierarchy of structures built upon the real numbers, which comprise only the set of real numbers (traditionally named ***R***), the set of complex numbers (traditionally named ***C***), the set of quaternions (traditionally named ***H***) and the set of octonions (traditionally named ***O***), which possess interesting mathematical properties (chief among which is the fact that they are *division algebras*, *i.e.* where the following property is true: if y is an element of that algebra and is **not equal to zero**, then $yx = yx'$, where x and x' denote elements of that algebra, implies that $x = x'$). Each member of the hierarchy is a super-set of the former.

One of the most important aspects of quaternions is that they provide an efficient way to parameterize rotations in $\boldsymbol{R^3}$ (the usual three-dimensional space) and $\boldsymbol{R^4}$.

In practical terms, a quaternion is simply a quadruple of real numbers (α,β,γ,δ), which we can write in the form $q = \quad + \quad i + \quad j + \quad k$, where i is the same object as for complex numbers, and j and k are distinct objects which play essentially the same kind of role as i.

An addition and a multiplication is defined on the set of quaternions, which generalize their real and complex counterparts. The main novelty here is that **the multiplication is not commutative** (i.e. there are quaternions x and y such that $xy \quad yx$). A good mnemotechnical way of remembering things is by using the formula $i*i = j*j = k*k = -1$.

Quaternions (and their kin) are described in far more details in this other document (with errata and addenda).

Some traditional constructs, such as the exponential, carry over without too much change into the realms of quaternions, but other, such as taking a square root, do not.

Header File

The interface and implementation are both supplied by the header file quaternion.hpp.

Synopsis

```
namespace boost{ namespace math{

template<typename T> class quaternion;
template<>           class quaternion<float>;
template<>           class quaternion<double>;
template<>           class quaternion<long double>;

// operators
template<typename T> quaternion<T> operator + (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator + (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator + (::std::complex<T> const & lhs, quaternion<T> const & rhs)
template<typename T> quaternion<T> operator + (quaternion<T> const & lhs, ::std::complex<T> const & rhs)
template<typename T> quaternion<T> operator + (quaternion<T> const & lhs, quaternion<T> const & rhs);

template<typename T> quaternion<T> operator - (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator - (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator - (::std::complex<T> const & lhs, quaternion<T> const & rhs)
template<typename T> quaternion<T> operator - (quaternion<T> const & lhs, ::std::complex<T> const & rhs)
template<typename T> quaternion<T> operator - (quaternion<T> const & lhs, quaternion<T> const & rhs);

template<typename T> quaternion<T> operator * (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator * (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator * (::std::complex<T> const & lhs, quaternion<T> const & rhs)
template<typename T> quaternion<T> operator * (quaternion<T> const & lhs, ::std::complex<T> const & rhs)
template<typename T> quaternion<T> operator * (quaternion<T> const & lhs, quaternion<T> const & rhs);

template<typename T> quaternion<T> operator / (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator / (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator / (::std::complex<T> const & lhs, quaternion<T> const & rhs)
template<typename T> quaternion<T> operator / (quaternion<T> const & lhs, ::std::complex<T> const & rhs)
template<typename T> quaternion<T> operator / (quaternion<T> const & lhs, quaternion<T> const & rhs);

template<typename T> quaternion<T> operator + (quaternion<T> const & q);
template<typename T> quaternion<T> operator - (quaternion<T> const & q);

template<typename T> bool operator == (T const & lhs, quaternion<T> const & rhs);
template<typename T> bool operator == (quaternion<T> const & lhs, T const & rhs);
template<typename T> bool operator == (::std::complex<T> const & lhs, quaternion<T> const & rhs);
template<typename T> bool operator == (quaternion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator == (quaternion<T> const & lhs, quaternion<T> const & rhs);

template<typename T> bool operator != (T const & lhs, quaternion<T> const & rhs);
template<typename T> bool operator != (quaternion<T> const & lhs, T const & rhs);
template<typename T> bool operator != (::std::complex<T> const & lhs, quaternion<T> const & rhs);
template<typename T> bool operator != (quaternion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator != (quaternion<T> const & lhs, quaternion<T> const & rhs);

template<typename T, typename charT, class traits>
::std::basic_istream<charT,traits>& operator >> (::std::basic_istream<charT,traits> & is, quaternion<T>

template<typename T, typename charT, class traits>
::std::basic_ostream<charT,traits>& operator operator << (::std::basic_ostream<charT,traits> & os, quate

// values
template<typename T> T              real(quaternion<T> const & q);
template<typename T> quaternion<T>  unreal(quaternion<T> const & q);

template<typename T> T              sup(quaternion<T> const & q);
template<typename T> T              l1(quaternion<T> const & q);
template<typename T> T              abs(quaternion<T> const & q);
template<typename T> T              norm(quaternion<T>const  & q);
template<typename T> quaternion<T>  conj(quaternion<T> const & q);
```

```
template<typename T> quaternion<T>  spherical(T const & rho, T const & theta, T const & phi1, T const &
template<typename T> quaternion<T>  semipolar(T const & rho, T const & alpha, T const & theta1, T const
template<typename T> quaternion<T>  multipolar(T const & rho1, T const & theta1, T const & rho2, T const
template<typename T> quaternion<T>  cylindrospherical(T const & t, T const & radius, T const & longitude
template<typename T> quaternion<T>  cylindrical(T const & r, T const & angle, T const & h1, T const & h2

// transcendentals
template<typename T> quaternion<T>  exp(quaternion<T> const & q);
template<typename T> quaternion<T>  cos(quaternion<T> const & q);
template<typename T> quaternion<T>  sin(quaternion<T> const & q);
template<typename T> quaternion<T>  tan(quaternion<T> const & q);
template<typename T> quaternion<T>  cosh(quaternion<T> const & q);
template<typename T> quaternion<T>  sinh(quaternion<T> const & q);
template<typename T> quaternion<T>  tanh(quaternion<T> const & q);
template<typename T> quaternion<T>  pow(quaternion<T> const & q, int n);

} // namespace math
} // namespace boost
```

Template Class quaternion

```
namespace boost{ namespace math{

template<typename T>
class quaternion
{
public:

   typedef T value_type;

   explicit quaternion(T const & requested_a = T(), T const & requested_b = T(), T const & requested_c =
   explicit quaternion(::std::complex<T> const & z0, ::std::complex<T> const & z1 = ::std::complex<T>())
   template<typename X>
   explicit quaternion(quaternion<X> const & a_recopier);

   T                  real() const;
   quaternion<T>      unreal() const;
   T                  R_component_1() const;
   T                  R_component_2() const;
   T                  R_component_3() const;
   T                  R_component_4() const;
   ::std::complex<T>  C_component_1() const;
   ::std::complex<T>  C_component_2() const;

   quaternion<T>&     operator = (quaternion<T> const  & a_affecter);
   template<typename X>
   quaternion<T>&     operator = (quaternion<X> const  & a_affecter);
   quaternion<T>&     operator = (T const  & a_affecter);
   quaternion<T>&     operator = (::std::complex<T> const & a_affecter);

   quaternion<T>&     operator += (T const & rhs);
   quaternion<T>&     operator += (::std::complex<T> const & rhs);
   template<typename X>
   quaternion<T>&     operator += (quaternion<X> const & rhs);

   quaternion<T>&     operator -= (T const & rhs);
   quaternion<T>&     operator -= (::std::complex<T> const & rhs);
   template<typename X>
   quaternion<T>&     operator -= (quaternion<X> const & rhs);

   quaternion<T>&     operator *= (T const & rhs);
   quaternion<T>&     operator *= (::std::complex<T> const & rhs);
   template<typename X>
   quaternion<T>&     operator *= (quaternion<X> const & rhs);

   quaternion<T>&     operator /= (T const & rhs);
   quaternion<T>&     operator /= (::std::complex<T> const & rhs);
   template<typename X>
   quaternion<T>&     operator /= (quaternion<X> const & rhs);
};

} // namespace math
} // namespace boost
```

Quaternion Specializations

```
namespace boost{ namespace math{

template<>
class quaternion<float>
{
public:
   typedef float value_type;

   explicit quaternion(float const & requested_a = 0.0f, float const & requested_b = 0.0f, float const &
   explicit quaternion(::std::complex<float> const & z0, ::std::complex<float> const & z1 = ::std::compl
   explicit quaternion(quaternion<double> const & a_recopier);
   explicit quaternion(quaternion<long double> const & a_recopier);

   float                   real() const;
   quaternion<float>       unreal() const;
   float                   R_component_1() const;
   float                   R_component_2() const;
   float                   R_component_3() const;
   float                   R_component_4() const;
   ::std::complex<float>   C_component_1() const;
   ::std::complex<float>   C_component_2() const;

   quaternion<float>&      operator = (quaternion<float> const  & a_affecter);
   template<typename X>
   quaternion<float>&      operator = (quaternion<X> const  & a_affecter);
   quaternion<float>&      operator = (float const  & a_affecter);
   quaternion<float>&      operator = (::std::complex<float> const & a_affecter);

   quaternion<float>&      operator += (float const & rhs);
   quaternion<float>&      operator += (::std::complex<float> const & rhs);
   template<typename X>
   quaternion<float>&      operator += (quaternion<X> const & rhs);

   quaternion<float>&      operator -= (float const & rhs);
   quaternion<float>&      operator -= (::std::complex<float> const & rhs);
   template<typename X>
   quaternion<float>&      operator -= (quaternion<X> const & rhs);

   quaternion<float>&      operator *= (float const & rhs);
   quaternion<float>&      operator *= (::std::complex<float> const & rhs);
   template<typename X>
   quaternion<float>&      operator *= (quaternion<X> const & rhs);

   quaternion<float>&      operator /= (float const & rhs);
   quaternion<float>&      operator /= (::std::complex<float> const & rhs);
   template<typename X>
   quaternion<float>&      operator /= (quaternion<X> const & rhs);
};
```

```
template<>
class quaternion<double>
{
public:
   typedef double value_type;

   explicit quaternion(double const & requested_a = 0.0, double const & requested_b = 0.0, double const
   explicit quaternion(::std::complex<double> const & z0, ::std::complex<double> const & z1 = ::std::com
   explicit quaternion(quaternion<float> const & a_recopier);
   explicit quaternion(quaternion<long double> const & a_recopier);

   double                  real() const;
   quaternion<double>      unreal() const;
   double                  R_component_1() const;
   double                  R_component_2() const;
   double                  R_component_3() const;
   double                  R_component_4() const;
   ::std::complex<double>  C_component_1() const;
   ::std::complex<double>  C_component_2() const;

   quaternion<double>&     operator = (quaternion<double> const  & a_affecter);
   template<typename X>
   quaternion<double>&     operator = (quaternion<X> const  & a_affecter);
   quaternion<double>&     operator = (double const  & a_affecter);
   quaternion<double>&     operator = (::std::complex<double> const & a_affecter);

   quaternion<double>&     operator += (double const & rhs);
   quaternion<double>&     operator += (::std::complex<double> const & rhs);
   template<typename X>
   quaternion<double>&     operator += (quaternion<X> const & rhs);

   quaternion<double>&     operator -= (double const & rhs);
   quaternion<double>&     operator -= (::std::complex<double> const & rhs);
   template<typename X>
   quaternion<double>&     operator -= (quaternion<X> const & rhs);

   quaternion<double>&     operator *= (double const & rhs);
   quaternion<double>&     operator *= (::std::complex<double> const & rhs);
   template<typename X>
   quaternion<double>&     operator *= (quaternion<X> const & rhs);

   quaternion<double>&     operator /= (double const & rhs);
   quaternion<double>&     operator /= (::std::complex<double> const & rhs);
   template<typename X>
   quaternion<double>&     operator /= (quaternion<X> const & rhs);
};
```

```
template<>
class quaternion<long double>
{
public:
   typedef long double value_type;

   explicit quaternion(long double const & requested_a = 0.0L, long double const & requested_b = 0.0L, l
   explicit quaternion(::std::complex<long double> const & z0, ::std::complex<long double> const & z1 =
   explicit quaternion(quaternion<float> const & a_recopier);
   explicit quaternion(quaternion<double> const & a_recopier);

   long double                  real() const;
   quaternion<long double>      unreal() const;
   long double                  R_component_1() const;
   long double                  R_component_2() const;
   long double                  R_component_3() const;
   long double                  R_component_4() const;
   ::std::complex<long double>  C_component_1() const;
   ::std::complex<long double>  C_component_2() const;

   quaternion<long double>&     operator = (quaternion<long double> const  & a_affecter);
   template<typename X>
   quaternion<long double>&     operator = (quaternion<X> const  & a_affecter);
   quaternion<long double>&     operator = (long double const  & a_affecter);
   quaternion<long double>&     operator = (::std::complex<long double> const & a_affecter);

   quaternion<long double>&     operator += (long double const & rhs);
   quaternion<long double>&     operator += (::std::complex<long double> const & rhs);
   template<typename X>
   quaternion<long double>&     operator += (quaternion<X> const & rhs);

   quaternion<long double>&     operator -= (long double const & rhs);
   quaternion<long double>&     operator -= (::std::complex<long double> const & rhs);
   template<typename X>
   quaternion<long double>&     operator -= (quaternion<X> const & rhs);

   quaternion<long double>&     operator *= (long double const & rhs);
   quaternion<long double>&     operator *= (::std::complex<long double> const & rhs);
   template<typename X>
   quaternion<long double>&     operator *= (quaternion<X> const & rhs);

   quaternion<long double>&     operator /= (long double const & rhs);
   quaternion<long double>&     operator /= (::std::complex<long double> const & rhs);
   template<typename X>
   quaternion<long double>&     operator /= (quaternion<X> const & rhs);
};

} // namespace math
} // namespace boost
```

Quaternion Member Typedefs

value_type

Template version:

```
typedef T value_type;
```

Float specialization version:

```
typedef float value_type;
```

Double specialization version:

```
typedef double value_type;
```

Long double specialization version:

```
typedef long double value_type;
```

These provide easy acces to the type the template is built upon.

Quaternion Member Functions

Constructors

Template version:

```
explicit quaternion(T const & requested_a = T(), T const & requested_b = T(), T const & requested_c = T(
explicit quaternion(::std::complex<T> const & z0, ::std::complex<T> const & z1 = ::std::complex<T>());
template<typename X>
explicit quaternion(quaternion<X> const & a_recopier);
```

Float specialization version:

```
explicit quaternion(float const & requested_a = 0.0f, float const & requested_b = 0.0f, float const & re
explicit quaternion(::std::complex<float> const & z0,::std::complex<float> const & z1 = ::std::complex<f
explicit quaternion(quaternion<double> const & a_recopier);
explicit quaternion(quaternion<long double> const & a_recopier);
```

Double specialization version:

```
explicit quaternion(double const & requested_a = 0.0, double const & requested_b = 0.0, double const & r
explicit quaternion(::std::complex<double> const & z0, ::std::complex<double> const & z1 = ::std::comple
explicit quaternion(quaternion<float> const & a_recopier);
explicit quaternion(quaternion<long double> const & a_recopier);
```

Long double specialization version:

```
explicit quaternion(long double const & requested_a = 0.0L, long double const & requested_b = 0.0L, long
explicit quaternion( ::std::complex<long double> const & z0, ::std::complex<long double> const & z1 = ::
explicit quaternion(quaternion<float> const & a_recopier);
explicit quaternion(quaternion<double> const & a_recopier);
```

A default constructor is provided for each form, which initializes each component to the default values for their type (i.e. zero for floating numbers). This constructor can also accept one to four base type arguments. A constructor is also provided to build quaternions from one or two complex numbers sharing the same base type. The unspecialized template also sports a templarized copy constructor,

while the specialized forms have copy constructors from the other two specializations, which are explicit when a risk of precision loss exists. For the unspecialized form, the base type's constructors must not throw.

Destructors and untemplated copy constructors (from the same type) are provided by the compiler. Converting copy constructors make use of a templated helper function in a "detail" subnamespace.

Other member functions

Real and Unreal Parts

```
T             real() const;
quaternion<T> unreal() const;
```

Like complex number, quaternions do have a meaningful notion of "real part", but unlike them there is no meaningful notion of "imaginary part". Instead there is an "unreal part" which itself is a quaternion, and usually nothing simpler (as opposed to the complex number case). These are returned by the first two functions.

Individual Real Components

```
T R_component_1() const;
T R_component_2() const;
T R_component_3() const;
T R_component_4() const;
```

A quaternion having four real components, these are returned by these four functions. Hence real and R_component_1 return the same value.

Individual Complex Components

```
::std::complex<T> C_component_1() const;
::std::complex<T> C_component_2() const;
```

A quaternion likewise has two complex components, and as we have seen above, for any quaternion $q = a + bi + cj + dk$ we also have $q = (a + bi) + (c + di)j$. These functions return them. The real part of `q.C_component_1()` is the same as `q.real()`.

Quaternion Member Operators

Assignment Operators

```
quaternion<T>& operator = (quaternion<T> const & a_affecter);
template<typename X>
quaternion<T>& operator = (quaternion<X> const& a_affecter);
quaternion<T>& operator = (T const& a_affecter);
quaternion<T>& operator = (::std::complex<T> const& a_affecter);
```

These perform the expected assignment, with type modification if necessary (for instance, assigning from a base type will set the real part to that value, and all other components to zero). For the unspecialized form, the base type's assignment operators must not throw.

Addition Operators

```
quaternion<T>& operator += (T const & rhs)
quaternion<T>& operator += (::std::complex<T> const & rhs);
template<typename X>
quaternion<T>& operator += (quaternion<X> const & rhs);
```

These perform the mathematical operation `(*this)+rhs` and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw.

Subtraction Operators

```
quaternion<T>& operator -= (T const & rhs)
quaternion<T>& operator -= (::std::complex<T> const & rhs);
template<typename X>
quaternion<T>& operator -= (quaternion<X> const & rhs);
```

These perform the mathematical operation `(*this)-rhs` and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw.

Multiplication Operators

```
quaternion<T>& operator *= (T const & rhs)
quaternion<T>& operator *= (::std::complex<T> const & rhs);
template<typename X>
quaternion<T>& operator *= (quaternion<X> const & rhs);
```

These perform the mathematical operation `(*this)*rhs` **in this order** (order is important as multiplication is not commutative for quaternions) and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw.

Division Operators

```
quaternion<T>& operator /= (T const & rhs)
quaternion<T>& operator /= (::std::complex<T> const & rhs);
template<typename X>
quaternion<T>& operator /= (quaternion<X> const & rhs);
```

These perform the mathematical operation `(*this)*inverse_of(rhs)` **in this order** (order is important as multiplication is not commutative for quaternions) and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw.

Quaternion Non-Member Operators

Unary Plus

```
template<typename T>
quaternion<T> operator + (quaternion<T> const & q);
```

This unary operator simply returns q.

Unary Minus

```
template<typename T>
quaternion<T> operator - (quaternion<T> const & q);
```

This unary operator returns the opposite of q.

Binary Addition Operators

```
template<typename T> quaternion<T> operator + (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator + (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator + (::std::complex<T> const & lhs, quaternion<T> const & rhs)
template<typename T> quaternion<T> operator + (quaternion<T> const & lhs, ::std::complex<T> const & rhs)
template<typename T> quaternion<T> operator + (quaternion<T> const & lhs, quaternion<T> const & rhs);
```

These operators return `quaternion<T>(lhs) += rhs`.

Binary Subtraction Operators

```
template<typename T> quaternion<T> operator - (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator - (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator - (::std::complex<T> const & lhs, quaternion<T> const & rhs)
template<typename T> quaternion<T> operator - (quaternion<T> const & lhs, ::std::complex<T> const & rhs)
template<typename T> quaternion<T> operator - (quaternion<T> const & lhs, quaternion<T> const & rhs);
```

These operators return `quaternion<T>(lhs) -= rhs`.

Binary Multiplication Operators

```
template<typename T> quaternion<T> operator * (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator * (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator * (::std::complex<T> const & lhs, quaternion<T> const & rhs)
template<typename T> quaternion<T> operator * (quaternion<T> const & lhs, ::std::complex<T> const & rhs)
template<typename T> quaternion<T> operator * (quaternion<T> const & lhs, quaternion<T> const & rhs);
```

These operators return `quaternion<T>(lhs) *= rhs`.

Binary Division Operators

```
template<typename T> quaternion<T> operator / (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator / (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator / (::std::complex<T> const & lhs, quaternion<T> const & rhs)
template<typename T> quaternion<T> operator / (quaternion<T> const & lhs, ::std::complex<T> const & rhs)
template<typename T> quaternion<T> operator / (quaternion<T> const & lhs, quaternion<T> const & rhs);
```

These operators return `quaternion<T>(lhs) /= rhs`. It is of course still an error to divide by zero...

Equality Operators

```
template<typename T> bool operator == (T const & lhs, quaternion<T> const & rhs);
template<typename T> bool operator == (quaternion<T> const & lhs, T const & rhs);
template<typename T> bool operator == (::std::complex<T> const & lhs, quaternion<T> const & rhs);
template<typename T> bool operator == (quaternion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator == (quaternion<T> const & lhs, quaternion<T> const & rhs);
```

These return true if and only if the four components of `quaternion<T>(lhs)` are equal to their counterparts in `quaternion<T>(rhs)`. As with any floating-type entity, this is essentially meaningless.

Inequality Operators

```
template<typename T> bool operator != (T const & lhs, quaternion<T> const & rhs);
template<typename T> bool operator != (quaternion<T> const & lhs, T const & rhs);
template<typename T> bool operator != (::std::complex<T> const & lhs, quaternion<T> const & rhs);
template<typename T> bool operator != (quaternion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator != (quaternion<T> const & lhs, quaternion<T> const & rhs);
```

These return true if and only if `quaternion<T>(lhs) == quaternion<T>(rhs)` is false. As with any floating-type entity, this is essentially meaningless.

Stream Extractor

```
template<typename T, typename charT, class traits>
::std::basic_istream<charT,traits>& operator >> (::std::basic_istream<charT,traits> & is, quaternion<T>
```

Extracts a quaternion q of one of the following forms (with a, b, c and d of type `T`):

`a (a), (a,b), (a,b,c), (a,b,c,d) (a,(c)), (a,(c,d)), ((a)), ((a),c), ((a),(c)), ((a),(c,d)), ((a,b)), ((a,b),c), ((a,b),(c)), ((a,b),(c,d))`

The input values must be convertible to `T`. If bad input is encountered, calls `is.setstate(ios::failbit)` (which may throw ios::failure (27.4.5.3)).

Returns: `is`.

The rationale for the list of accepted formats is that either we have a list of up to four reals, or else we have a couple of complex numbers, and in that case if it formated as a proper complex number, then it should be accepted. Thus potential ambiguities are lifted (for instance (a,b) is (a,b,0,0) and not (a,0,b,0), i.e. it is parsed as a list of two real numbers and not two complex numbers which happen to have imaginary parts equal to zero).

Stream Inserter

```
template<typename T, typename charT, class traits>
::std::basic_ostream<charT,traits>& operator << (::std::basic_ostream<charT,traits> & os, quaternion<T>
```

Inserts the quaternion q onto the stream `os` as if it were implemented as follows:

```
template<typename T, typename charT, class traits>
::std::basic_ostream<charT,traits>& operator << (
               ::std::basic_ostream<charT,traits> & os,
               quaternion<T> const & q)
{
   ::std::basic_ostringstream<charT,traits> s;

   s.flags(os.flags());
   s.imbue(os.getloc());
   s.precision(os.precision());

   s << '(' << q.R_component_1() << ','
            << q.R_component_2() << ','
            << q.R_component_3() << ','
            << q.R_component_4() << ')';

   return os << s.str();
}
```

Quaternion Value Operations

real and unreal

```
template<typename T> T             real(quaternion<T> const & q);
template<typename T> quaternion<T>  unreal(quaternion<T> const & q);
```

These return `q.real()` and `q.unreal()` respectively.

conj

```
template<typename T> quaternion<T>  conj(quaternion<T> const & q);
```

This returns the conjugate of the quaternion.

sup

template<typename T> T sup(quaternion<T> const & q);

This return the sup norm (the greatest among `abs(q.R_component_1())...abs(q.R_component_4())`) of the quaternion.

l1

```
template<typename T> T  l1(quaternion<T> const & q);
```

This return the l1 norm `(abs(q.R_component_1())+...+abs(q.R_component_4()))` of the quaternion.

abs

```
template<typename T> T  abs(quaternion<T> const & q);
```

This return the magnitude (Euclidian norm) of the quaternion.

norm

```
template<typename T> T  norm(quaternion<T>const  & q);
```

This return the (Cayley) norm of the quaternion. The term "norm" might be confusing, as most people associate it with the Euclidian norm (and quadratic functionals). For this version of (the mathematical objects known as) quaternions, the Euclidian norm (also known as magnitude) is the square root of the Cayley norm.

Quaternion Creation Functions

```
template<typename T> quaternion<T> spherical(T const & rho, T const & theta, T const & phi1, T const & p
template<typename T> quaternion<T> semipolar(T const & rho, T const & alpha, T const & theta1, T const &
template<typename T> quaternion<T> multipolar(T const & rho1, T const & theta1, T const & rho2, T const
template<typename T> quaternion<T> cylindrospherical(T const & t, T const & radius, T const & longitude,
template<typename T> quaternion<T> cylindrical(T const & r, T const & angle, T const & h1, T const & h2)
```

These build quaternions in a way similar to the way polar builds complex numbers, as there is no strict equivalent to polar coordinates for quaternions.

`spherical` is a simple transposition of `polar`, it takes as inputs a (positive) magnitude and a point on the hypersphere, given by three angles. The first of these, `theta` has a natural range of `-pi` to `+pi`, and the other two have natural ranges of `-pi/2` to `+pi/2` (as is the case with the usual spherical coordinates in $\boldsymbol{R^3}$). Due to the many symmetries and periodicities, nothing untoward happens if the magnitude is negative or the angles are outside their natural ranges. The expected degeneracies (a magnitude of zero ignores the angles settings...) do happen however.

`cylindrical` is likewise a simple transposition of the usual cylindrical coordinates in $\boldsymbol{R^3}$, which in turn is another derivative of planar polar coordinates. The first two inputs are the polar coordinates of the first $\boldsymbol{C}$ component of the quaternion. The third and fourth inputs are placed into the third and fourth $\boldsymbol{R}$ components of the quaternion, respectively.

`multipolar` is yet another simple generalization of polar coordinates. This time, both $\boldsymbol{C}$ components of the quaternion are given in polar coordinates.

`cylindrospherical` is specific to quaternions. It is often interesting to consider $\boldsymbol{H}$ as the cartesian product of $\boldsymbol{R}$ by $\boldsymbol{R^3}$ (the quaternionic multiplication as then a special form, as given here). This function therefore builds a quaternion from this representation, with the $\boldsymbol{R^3}$ component given in usual $\boldsymbol{R^3}$ spherical coordinates.

`semipolar` is another generator which is specific to quaternions. It takes as a first input the magnitude of the quaternion, as a second input an angle in the range `0` to `+pi/2` such that magnitudes of the first two $\boldsymbol{C}$ components of the quaternion are the product of the first input and the sine and cosine of this angle, respectively, and finally as third and fourth inputs angles in the range `-pi/2` to `+pi/2` which represent the arguments of the first and second $\boldsymbol{C}$ components of the quaternion, respectively. As usual, nothing untoward happens if what should be magnitudes are negative numbers or angles are out of their natural ranges, as symmetries and periodicities kick in.

In this version of our implementation of quaternions, there is no analogue of the complex value operation `arg` as the situation is somewhat more complicated. Unit quaternions are linked both to rotations in $\boldsymbol{R^3}$ and in $\boldsymbol{R^4}$, and the correspondences are not too complicated, but there is currently a lack of standard (de facto or de jure) matrix library with which the conversions could work. This should be remedied in a further revision. In the mean time, an example of how this could be done is presented here for $\boldsymbol{R^3}$, and here for $\boldsymbol{R^4}$ (example test file).

Quaternion Transcendentals

There is no `log` or `sqrt` provided for quaternions in this implementation, and `pow` is likewise restricted to integral powers of the exponent. There are several reasons to this: on the one hand, the equivalent of analytic continuation for quaternions ("branch cuts") remains to be investigated thoroughly (by me, at any rate...), and we wish to avoid the nonsense introduced in the standard by expo-

nentiations of complexes by complexes (which is well defined, but not in the standard...). Talking of nonsense, saying that `pow(0,0)` is "implementation defined" is just plain brain-dead...

We do, however provide several transcendentals, chief among which is the exponential. This author claims the complete proof of the "closed formula" as his own, as well as its independant invention (there are claims to prior invention of the formula, such as one by Professor Shoemake, and it is possible that the formula had been known a couple of centuries back, but in absence of bibliographical reference, the matter is pending, awaiting further investigation; on the other hand, the definition and existence of the exponential on the quaternions, is of course a fact known for a very long time). Basically, any converging power series with real coefficients which allows for a closed formula in ***C*** can be transposed to ***H***. More transcendentals of this type could be added in a further revision upon request. It should be noted that it is these functions which force the dependency upon the boost/math/special_functions/sinc.hpp and the boost/math/special_functions/sinhc.hpp headers.

exp

```
template<typename T> quaternion<T> exp(quaternion<T> const & q);
```

Computes the exponential of the quaternion.

cos

```
template<typename T> quaternion<T>  cos(quaternion<T> const & q);
```

Computes the cosine of the quaternion

sin

```
template<typename T> quaternion<T>  sin(quaternion<T> const & q);
```

Computes the sine of the quaternion.

tan

```
template<typename T> quaternion<T>  tan(quaternion<T> const & q);
```

Computes the tangent of the quaternion.

cosh

```
template<typename T> quaternion<T>  cosh(quaternion<T> const & q);
```

Computes the hyperbolic cosine of the quaternion.

sinh

```
template<typename T> quaternion<T>  sinh(quaternion<T> const & q);
```

Computes the hyperbolic sine of the quaternion.

tanh

```
template<typename T> quaternion<T>  tanh(quaternion<T> const & q);
```

Computes the hyperbolic tangent of the quaternion.

pow

```
template<typename T> quaternion<T>  pow(quaternion<T> const & q, int n);
```

Computes the n-th power of the quaternion q.

Test Program

The quaternion_test.cpp test program tests quaternions specializations for float, double and long double (sample output, with message output enabled).

If you define the symbol BOOST_QUATERNION_TEST_VERBOSE, you will get additional output (verbose output); this will only be helpfull if you enable message output at the same time, of course (by uncommenting the relevant line in the test or by adding `--log_level=messages` to your command line,...). In that case, and if you are running interactively, you may in addition define the symbol BOOST_INTERACTIVE_TEST_INPUT_ITERATOR to interactively test the input operator with input of your choice from the standard input (instead of hard-coding it in the test).

Acknowledgements

The mathematical text has been typeset with Nisus Writer. Jens Maurer has helped with portability and standard adherence, and was the Review Manager for this library. More acknowledgements in the History section. Thank you to all who contributed to the discution about this library.

History

- 1.5.8 - 17/12/2005: Converted documentation to Quickbook Format.
- 1.5.7 - 24/02/2003: transitionned to the unit test framework; <boost/config.hpp> now included by the library header (rather than the test files).
- 1.5.6 - 15/10/2002: Gcc2.95.x and stlport on linux compatibility by Alkis Evlogimenos (alkis@routescience.com).
- 1.5.5 - 27/09/2002: Microsoft VCPP 7 compatibility, by Michael Stevens (michael@acfr.usyd.edu.au); requires the /Za compiler option.
- 1.5.4 - 19/09/2002: fixed problem with multiple inclusion (in different translation units); attempt at an improved compatibility with Microsoft compilers, by Michael Stevens (michael@acfr.usyd.edu.au) and Fredrik Blomqvist; other compatibility fixes.
- 1.5.3 - 01/02/2002: bugfix and Gcc 2.95.3 compatibility by Douglas Gregor (gregod@cs.rpi.edu).
- 1.5.2 - 07/07/2001: introduced namespace math.
- 1.5.1 - 07/06/2001: (end of Boost review) now includes <boost/math/special_functions/sinc.hpp> and <boost/math/special_functions/sinhc.hpp> instead of <boost/special_functions.hpp>; corrected bug in sin (Daryle Walker); removed check for self-assignment (Gary Powel); made converting functions explicit (Gary Powel); added overflow guards for division operators and abs (Peter Schmitteckert); added sup and l1; used Vesa Karvonen's CPP metaprograming technique to simplify code.
- 1.5.0 - 26/03/2001: boostification, inlining of all operators except input, output and pow, fixed exception safety of some members (template version) and output operator, added spherical, semipolar, multipolar, cylindrospherical and cylindrical.

- 1.4.0 - 09/01/2001: added tan and tanh.
- 1.3.1 - 08/01/2001: cosmetic fixes.
- 1.3.0 - 12/07/2000: pow now uses Maarten Hilferink's (mhilferink@tip.nl) algorithm.
- 1.2.0 - 25/05/2000: fixed the division operators and output; changed many signatures.
- 1.1.0 - 23/05/2000: changed sinc into sinc_pi; added sin, cos, sinh, cosh.
- 1.0.0 - 10/08/1999: first public version.

To Do

- Improve testing.
- Rewrite input operatore using Spirit (creates a dependency).
- Put in place an Expression Template mechanism (perhaps borrowing from uBlas).
- Use uBlas for the link with rotations (and move from the example implementation to an efficient one).

Octonions

Overview

Octonions, like quaternions, are a relative of complex numbers.

Octonions see some use in theoretical physics.

In practical terms, an octonion is simply an octuple of real numbers (α,β,γ,δ,ε,ζ,η,θ), which we can write in the form $o = + i + j + k + e' + i' + j' + k'$, where i, j and k are the same objects as for quaternions, and e', i', j' and k' are distinct objects which play essentially the same kind of role as i (or j or k).

Addition and a multiplication is defined on the set of octonions, which generalize their quaternionic counterparts. The main novelty this time is that **the multiplication is not only not commutative, is now not even associative** (i.e. there are quaternions x, y and z such that $x(yz) \neq (xy)z$). A way of remembering things is by using the following multiplication table:

	1	i	j	k	e'	i'	j'	k'
1	1	i	j	k	e'	i'	j'	k'
i	i	-1	k	$-j$	i'	$-e'$	$-k'$	j'
j	j	$-k$	-1	i	j'	k'	$-e'$	$-i'$
k	k	j	$-i$	-1	k'	$-j'$	i'	$-e'$
e'	e'	$-i'$	$-j'$	$-k'$	-1	i	j	k
i'	i'	e'	$-k'$	j'	$-i$	-1	$-k$	j
j'	j'	k'	e'	$-i'$	$-j$	k	-1	$-i$
k'	k'	$-j'$	i'	e'	$-k$	$-j$	i	-1

Octonions (and their kin) are described in far more details in this other document (with errata and addenda).

Some traditional constructs, such as the exponential, carry over without too much change into the realms of octonions, but other, such as taking a square root, do not (the fact that the exponential has a closed form is a result of the author, but the fact that the exponential exists at all for octonions is known since quite a long time ago).

Header File

The interface and implementation are both supplied by the header file octonion.hpp.

Synopsis

```
namespace boost{ namespace math{

template<typename T> class octonion;
template<>           class octonion<float>;
template<>           class octonion<double>;
template<>           class octonion<long double>;

// operators

template<typename T> octonion<T> operator + (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator + (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator + (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator + (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> octonion<T> operator + (::boost::math::quaternion<T> const & lhs, octonion<T> const
template<typename T> octonion<T> operator + (octonion<T> const & lhs, ::boost::math::quaternion<T> const
template<typename T> octonion<T> operator + (octonion<T> const & lhs, octonion<T> const & rhs);

template<typename T> octonion<T> operator - (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator - (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator - (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator - (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> octonion<T> operator - (::boost::math::quaternion<T> const & lhs, octonion<T> const
template<typename T> octonion<T> operator - (octonion<T> const & lhs, ::boost::math::quaternion<T> const
template<typename T> octonion<T> operator - (octonion<T> const & lhs, octonion<T> const & rhs);

template<typename T> octonion<T> operator * (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator * (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator * (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator * (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> octonion<T> operator * (::boost::math::quaternion<T> const & lhs, octonion<T> const
template<typename T> octonion<T> operator * (octonion<T> const & lhs, ::boost::math::quaternion<T> const
template<typename T> octonion<T> operator * (octonion<T> const & lhs, octonion<T> const & rhs);

template<typename T> octonion<T> operator / (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator / (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator / (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator / (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> octonion<T> operator / (::boost::math::quaternion<T> const & lhs, octonion<T> const
template<typename T> octonion<T> operator / (octonion<T> const & lhs, ::boost::math::quaternion<T> const
template<typename T> octonion<T> operator / (octonion<T> const & lhs, octonion<T> const & rhs);

template<typename T> octonion<T> operator + (octonion<T> const & o);
template<typename T> octonion<T> operator - (octonion<T> const & o);

template<typename T> bool operator == (T const & lhs, octonion<T> const & rhs);
template<typename T> bool operator == (octonion<T> const & lhs, T const & rhs);
template<typename T> bool operator == (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> bool operator == (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator == (::boost::math::quaternion<T> const & lhs, octonion<T> const & rhs
template<typename T> bool operator == (octonion<T> const & lhs, ::boost::math::quaternion<T> const & rhs
template<typename T> bool operator == (octonion<T> const & lhs, octonion<T> const & rhs);

template<typename T> bool operator != (T const & lhs, octonion<T> const & rhs);
template<typename T> bool operator != (octonion<T> const & lhs, T const & rhs);
template<typename T> bool operator != (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> bool operator != (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator != (::boost::math::quaternion<T> const & lhs, octonion<T> const & rhs
template<typename T> bool operator != (octonion<T> const & lhs, ::boost::math::quaternion<T> const & rhs
template<typename T> bool operator != (octonion<T> const & lhs, octonion<T> const & rhs);

template<typename T, typename charT, class traits>
::std::basic_istream<charT,traits> & operator >> (::std::basic_istream<charT,traits> & is, octonion<T> &
```

```
template<typename T, typename charT, class traits>
::std::basic_ostream<charT,traits> & operator << (::std::basic_ostream<charT,traits> & os, octonion<T> c

// values

template<typename T> T           real(octonion<T> const & o);
template<typename T> octonion<T> unreal(octonion<T> const & o);

template<typename T> T           sup(octonion<T> const & o);
template<typename T> T           l1(octonion<T>const & o);
template<typename T> T           abs(octonion<T> const & o);
template<typename T> T           norm(octonion<T>const  & o);
template<typename T> octonion<T> conj(octonion<T> const & o);

template<typename T> octonion<T> spherical(T const & rho, T const & theta, T const & phi1, T const & phi
template<typename T> octonion<T> multipolar(T const & rho1, T const & theta1, T const & rho2, T const &
template<typename T> octonion<T> cylindrical(T const & r, T const & angle, T const & h1, T const & h2, T

// transcendentals

template<typename T> octonion<T> exp(octonion<T> const & o);
template<typename T> octonion<T> cos(octonion<T> const & o);
template<typename T> octonion<T> sin(octonion<T> const & o);
template<typename T> octonion<T> tan(octonion<T> const & o);
template<typename T> octonion<T> cosh(octonion<T> const & o);
template<typename T> octonion<T> sinh(octonion<T> const & o);
template<typename T> octonion<T> tanh(octonion<T> const & o);

template<typename T> octonion<T> pow(octonion<T> const & o, int n);

}  } // namespaces
```

Template Class octonion

```
namespace boost{ namespace math {

template<typename T>
class octonion
{
public:
   typedef T value_type;

   explicit  octonion(T const & requested_a = T(), T const & requested_b = T(), T const & requested_c =
   explicit  octonion(::std::complex<T> const & z0, ::std::complex<T> const & z1 = ::std::complex<T>(),
   explicit  octonion(::boost::math::quaternion<T> const & q0, ::boost::math::quaternion<T> const & q1 =
   template<typename X>
   explicit  octonion(octonion<X> const & a_recopier);

   T                              real() const;
   octonion<T>                    unreal() const;

   T                              R_component_1() const;
   T                              R_component_2() const;
   T                              R_component_3() const;
   T                              R_component_4() const;
   T                              R_component_5() const;
   T                              R_component_6() const;
   T                              R_component_7() const;
   T                              R_component_8() const;

   ::std::complex<T>              C_component_1() const;
   ::std::complex<T>              C_component_2() const;
   ::std::complex<T>              C_component_3() const;
   ::std::complex<T>              C_component_4() const;

   ::boost::math::quaternion<T>   H_component_1() const;
   ::boost::math::quaternion<T>   H_component_2() const;

   octonion<T> & operator = (octonion<T> const  & a_affecter);
   template<typename X>
   octonion<T> & operator = (octonion<X> const  & a_affecter);
   octonion<T> & operator = (T const  & a_affecter);
   octonion<T> & operator = (::std::complex<T> const & a_affecter);
   octonion<T> & operator = (::boost::math::quaternion<T> const & a_affecter);

   octonion<T> & operator += (T const & rhs);
   octonion<T> & operator += (::std::complex<T> const & rhs);
   octonion<T> & operator += (::boost::math::quaternion<T> const & rhs);
   template<typename X>
   octonion<T> & operator += (octonion<X> const & rhs);

   octonion<T> & operator -= (T const & rhs);
   octonion<T> & operator -= (::std::complex<T> const & rhs);
   octonion<T> & operator -= (::boost::math::quaternion<T> const & rhs);
   template<typename X>
   octonion<T> & operator -= (octonion<X> const & rhs);

   octonion<T> & operator *= (T const & rhs);
   octonion<T> & operator *= (::std::complex<T> const & rhs);
   octonion<T> & operator *= (::boost::math::quaternion<T> const & rhs);
   template<typename X>
   octonion<T> & operator *= (octonion<X> const & rhs);

   octonion<T> & operator /= (T const & rhs);
   octonion<T> & operator /= (::std::complex<T> const & rhs);
   octonion<T> & operator /= (::boost::math::quaternion<T> const & rhs);
```

```
   template<typename X>
   octonion<T> & operator /= (octonion<X> const & rhs);
};

} } // namespaces
```

Octonion Specializations

```
namespace boost{ namespace math{

template<>
class octonion<float>
{
public:
   typedef float value_type;

   explicit  octonion(float const & requested_a = 0.0f, float const & requested_b = 0.0f, float const &
   explicit  octonion(::std::complex<float> const & z0, ::std::complex<float> const & z1 = ::std::comple
   explicit  octonion(::boost::math::quaternion<float> const & q0, ::boost::math::quaternion<float> cons
   explicit  octonion(octonion<double> const & a_recopier);
   explicit  octonion(octonion<long double> const & a_recopier);

   float                              real() const;
   octonion<float>                    unreal() const;

   float                              R_component_1() const;
   float                              R_component_2() const;
   float                              R_component_3() const;
   float                              R_component_4() const;
   float                              R_component_5() const;
   float                              R_component_6() const;
   float                              R_component_7() const;
   float                              R_component_8() const;

   ::std::complex<float>              C_component_1() const;
   ::std::complex<float>              C_component_2() const;
   ::std::complex<float>              C_component_3() const;
   ::std::complex<float>              C_component_4() const;

   ::boost::math::quaternion<float>   H_component_1() const;
   ::boost::math::quaternion<float>   H_component_2() const;

   octonion<float> & operator = (octonion<float> const  & a_affecter);
   template<typename X>
   octonion<float> & operator = (octonion<X> const  & a_affecter);
   octonion<float> & operator = (float const  & a_affecter);
   octonion<float> & operator = (::std::complex<float> const & a_affecter);
   octonion<float> & operator = (::boost::math::quaternion<float> const & a_affecter);

   octonion<float> & operator += (float const & rhs);
   octonion<float> & operator += (::std::complex<float> const & rhs);
   octonion<float> & operator += (::boost::math::quaternion<float> const & rhs);
   template<typename X>
   octonion<float> & operator += (octonion<X> const & rhs);

   octonion<float> & operator -= (float const & rhs);
   octonion<float> & operator -= (::std::complex<float> const & rhs);
   octonion<float> & operator -= (::boost::math::quaternion<float> const & rhs);
   template<typename X>
   octonion<float> & operator -= (octonion<X> const & rhs);

   octonion<float> & operator *= (float const & rhs);
   octonion<float> & operator *= (::std::complex<float> const & rhs);
   octonion<float> & operator *= (::boost::math::quaternion<float> const & rhs);
   template<typename X>
   octonion<float> & operator *= (octonion<X> const & rhs);

   octonion<float> & operator /= (float const & rhs);
   octonion<float> & operator /= (::std::complex<float> const & rhs);
   octonion<float> & operator /= (::boost::math::quaternion<float> const & rhs);
```

```
   template<typename X>
   octonion<float> & operator /= (octonion<X> const & rhs);
};
```

```
template<>
class octonion<double>
{
public:
   typedef double value_type;

   explicit  octonion(double const & requested_a = 0.0, double const & requested_b = 0.0, double const &
   explicit  octonion(::std::complex<double> const & z0, ::std::complex<double> const & z1 = ::std::comp
   explicit  octonion(::boost::math::quaternion<double> const & q0, ::boost::math::quaternion<double> co
   explicit  octonion(octonion<float> const & a_recopier);
   explicit  octonion(octonion<long double> const & a_recopier);

   double                              real() const;
   octonion<double>                    unreal() const;

   double                              R_component_1() const;
   double                              R_component_2() const;
   double                              R_component_3() const;
   double                              R_component_4() const;
   double                              R_component_5() const;
   double                              R_component_6() const;
   double                              R_component_7() const;
   double                              R_component_8() const;

   ::std::complex<double>              C_component_1() const;
   ::std::complex<double>              C_component_2() const;
   ::std::complex<double>              C_component_3() const;
   ::std::complex<double>              C_component_4() const;

   ::boost::math::quaternion<double>   H_component_1() const;
   ::boost::math::quaternion<double>   H_component_2() const;

   octonion<double> & operator = (octonion<double> const  & a_affecter);
   template<typename X>
   octonion<double> & operator = (octonion<X> const  & a_affecter);
   octonion<double> & operator = (double const  & a_affecter);
   octonion<double> & operator = (::std::complex<double> const & a_affecter);
   octonion<double> & operator = (::boost::math::quaternion<double> const & a_affecter);

   octonion<double> & operator += (double const & rhs);
   octonion<double> & operator += (::std::complex<double> const & rhs);
   octonion<double> & operator += (::boost::math::quaternion<double> const & rhs);
   template<typename X>
   octonion<double> & operator += (octonion<X> const & rhs);

   octonion<double> & operator -= (double const & rhs);
   octonion<double> & operator -= (::std::complex<double> const & rhs);
   octonion<double> & operator -= (::boost::math::quaternion<double> const & rhs);
   template<typename X>
   octonion<double> & operator -= (octonion<X> const & rhs);

   octonion<double> & operator *= (double const & rhs);
   octonion<double> & operator *= (::std::complex<double> const & rhs);
   octonion<double> & operator *= (::boost::math::quaternion<double> const & rhs);
   template<typename X>
   octonion<double> & operator *= (octonion<X> const & rhs);

   octonion<double> & operator /= (double const & rhs);
   octonion<double> & operator /= (::std::complex<double> const & rhs);
   octonion<double> & operator /= (::boost::math::quaternion<double> const & rhs);
   template<typename X>
```

```
   octonion<double> & operator /= (octonion<X> const & rhs);
};
```

```
template<>
class octonion<long double>
{
public:
   typedef long double value_type;

   explicit   octonion(long double const & requested_a = 0.0L, long double const & requested_b = 0.0L, l
   explicit   octonion( ::std::complex<long double> const & z0, ::std::complex<long double> const & z1 =
   explicit   octonion( ::boost::math::quaternion<long double> const & q0, ::boost::math::quaternion<lor
   explicit   octonion(octonion<float> const & a_recopier);
   explicit   octonion(octonion<double> const & a_recopier);

   long double                              real() const;
   octonion<long double>                    unreal() const;

   long double                              R_component_1() const;
   long double                              R_component_2() const;
   long double                              R_component_3() const;
   long double                              R_component_4() const;
   long double                              R_component_5() const;
   long double                              R_component_6() const;
   long double                              R_component_7() const;
   long double                              R_component_8() const;

   ::std::complex<long double>              C_component_1() const;
   ::std::complex<long double>              C_component_2() const;
   ::std::complex<long double>              C_component_3() const;
   ::std::complex<long double>              C_component_4() const;

   ::boost::math::quaternion<long double>   H_component_1() const;
   ::boost::math::quaternion<long double>   H_component_2() const;

   octonion<long double> & operator = (octonion<long double> const  & a_affecter);
   template<typename X>
   octonion<long double> & operator = (octonion<X> const  & a_affecter);
   octonion<long double> & operator = (long double const  & a_affecter);
   octonion<long double> & operator = (::std::complex<long double> const & a_affecter);
   octonion<long double> & operator = (::boost::math::quaternion<long double> const & a_affecter);

   octonion<long double> & operator += (long double const & rhs);
   octonion<long double> & operator += (::std::complex<long double> const & rhs);
   octonion<long double> & operator += (::boost::math::quaternion<long double> const & rhs);
   template<typename X>
   octonion<long double> & operator += (octonion<X> const & rhs);

   octonion<long double> & operator -= (long double const & rhs);
   octonion<long double> & operator -= (::std::complex<long double> const & rhs);
   octonion<long double> & operator -= (::boost::math::quaternion<long double> const & rhs);
   template<typename X>
   octonion<long double> & operator -= (octonion<X> const & rhs);

   octonion<long double> & operator *= (long double const & rhs);
   octonion<long double> & operator *= (::std::complex<long double> const & rhs);
   octonion<long double> & operator *= (::boost::math::quaternion<long double> const & rhs);
   template<typename X>
   octonion<long double> & operator *= (octonion<X> const & rhs);

   octonion<long double> & operator /= (long double const & rhs);
   octonion<long double> & operator /= (::std::complex<long double> const & rhs);
   octonion<long double> & operator /= (::boost::math::quaternion<long double> const & rhs);
   template<typename X>
   octonion<long double> & operator /= (octonion<X> const & rhs);
```

```
};

} } // namespaces
```

Octonion Member Typedefs

value_type

Template version:

```
typedef T value_type;
```

Float specialization version:

```
typedef float value_type;
```

Double specialization version:

```
typedef double value_type;
```

Long double specialization version:

```
typedef long double value_type;
```

These provide easy acces to the type the template is built upon.

Octonion Member Functions

Constructors

Template version:

```
explicit  octonion(T const & requested_a = T(), T const & requested_b = T(), T const & requested_c = T()
explicit  octonion(::std::complex<T> const & z0, ::std::complex<T> const & z1 = ::std::complex<T>(), ::s
explicit  octonion(::boost::math::quaternion<T> const & q0, ::boost::math::quaternion<T> const & q1 = ::
template<typename X>
explicit octonion(octonion<X> const & a_recopier);
```

Float specialization version:

```
explicit  octonion(float const & requested_a = 0.0f, float const & requested_b = 0.0f, float const & req
explicit  octonion(::std::complex<float> const & z0, ::std::complex<float> const & z1 = ::std::complex<f
explicit  octonion(::boost::math::quaternion<float> const & q0, ::boost::math::quaternion<float> const &
explicit  octonion(octonion<double> const & a_recopier);
explicit  octonion(octonion<long double> const & a_recopier);
```

Double specialization version:

```
explicit  octonion(double const & requested_a = 0.0, double const & requested_b = 0.0, double const & re
explicit  octonion(::std::complex<double> const & z0, ::std::complex<double> const & z1 = ::std::complex
explicit  octonion(::boost::math::quaternion<double> const & q0, ::boost::math::quaternion<double> const
explicit  octonion(octonion<float> const & a_recopier);
explicit  octonion(octonion<long double> const & a_recopier);
```

Long double specialization version:

```
explicit  octonion(long double const & requested_a = 0.0L, long double const & requested_b = 0.0L, long
explicit  octonion( ::std::complex<long double> const & z0, ::std::complex<long double> const & z1 = ::s
explicit  octonion(::boost::math::quaternion<long double> const & q0, ::boost::math::quaternion<long dou
explicit  octonion(octonion<float> const & a_recopier);
explicit  octonion(octonion<double> const & a_recopier);
```

A default constructor is provided for each form, which initializes each component to the default values for their type (i.e. zero for floating numbers). This constructor can also accept one to eight base type arguments. A constructor is also provided to build octonions from one to four complex numbers sharing the same base type, and another taking one or two quaternions sharing the same base type. The unspecialized template also sports a templarized copy constructor, while the specialized forms have copy constructors from the other two specializations, which are explicit when a risk of precision loss exists. For the unspecialized form, the base type's constructors must not throw.

Destructors and untemplated copy constructors (from the same type) are provided by the compiler. Converting copy constructors make use of a templated helper function in a "detail" subnamespace.

Other member functions

Real and Unreal Parts

```
T           real()   const;
octonion<T> unreal() const;
```

Like complex number, octonions do have a meaningful notion of "real part", but unlike them there is no meaningful notion of "imaginary part". Instead there is an "unreal part" which itself is a octonion, and usually nothing simpler (as opposed to the complex number case). These are returned by the first two functions.

Individual Real Components

```
T   R_component_1() const;
T   R_component_2() const;
T   R_component_3() const;
T   R_component_4() const;
T   R_component_5() const;
T   R_component_6() const;
T   R_component_7() const;
T   R_component_8() const;
```

A octonion having eight real components, these are returned by these eight functions. Hence real and R_component_1 return the same value.

Individual Complex Components

```
::std::complex<T> C_component_1() const;
::std::complex<T> C_component_2() const;
::std::complex<T> C_component_3() const;
::std::complex<T> C_component_4() const;
```

A octonion likewise has four complex components. Actually, octonions are indeed a (left) vector field over the complexes, but beware, as for any octonion *o = + i + j + k + e' + i' + j' + k'* we also have *o = (+ i) + (+ i)j + (+ i)e' + (- i)j'* (note the **minus** sign in the last factor). What the C_component_n functions return, however, are the complexes which could be used to build the octonion using the constructor, and **not** the components of the octonion on the basis *(1, j, e', j')*.

Individual Quaternion Components

```
::boost::math::quaternion<T> H_component_1() const;
::boost::math::quaternion<T> H_component_2() const;
```

Likewise, for any octonion *o = + i + j + k + e' + i' + j' + k'* we also have *o = (+ i + j + k) + (+ i + j - j)e'*, though there is no meaningful vector-space-like structure based on the quaternions. What the H_component_n functions return are the quaternions which could be used to build the octonion using the constructor.

Octonion Member Operators

Assignment Operators

```
octonion<T> & operator = (octonion<T> const & a_affecter);
template<typename X>
octonion<T> & operator = (octonion<X> const & a_affecter);
octonion<T> & operator = (T const & a_affecter);
octonion<T> & operator = (::std::complex<T> const & a_affecter);
octonion<T> & operator = (::boost::math::quaternion<T> const & a_affecter);
```

These perform the expected assignment, with type modification if necessary (for instance, assigning from a base type will set the real part to that value, and all other components to zero). For the unspecialized form, the base type's assignment operators must not throw.

Other Member Operators

```
octonion<T> & operator += (T const & rhs)
octonion<T> & operator += (::std::complex<T> const & rhs);
octonion<T> & operator += (::boost::math::quaternion<T> const & rhs);
template<typename X>
octonion<T> & operator += (octonion<X> const & rhs);
```

These perform the mathematical operation `(*this)+rhs` and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw.

```
octonion<T> & operator -= (T const & rhs)
octonion<T> & operator -= (::std::complex<T> const & rhs);
octonion<T> & operator -= (::boost::math::quaternion<T> const & rhs);
template<typename X>
octonion<T> & operator -= (octonion<X> const & rhs);
```

These perform the mathematical operation `(*this)-rhs` and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw.

```
octonion<T> & operator *= (T const & rhs)
octonion<T> & operator *= (::std::complex<T> const & rhs);
octonion<T> & operator *= (::boost::math::quaternion<T> const & rhs);
template<typename X>
octonion<T> & operator *= (octonion<X> const & rhs);
```

These perform the mathematical operation `(*this)*rhs` in this order (order is important as multiplication is not commutative for octonions) and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw. Also, for clarity's sake, you should always group the factors in a multiplication by groups of two, as the multiplication is not even associative on the octonions (though there are of course cases where this does not matter, it usually does).

```
octonion<T> & operator /= (T const & rhs)
octonion<T> & operator /= (::std::complex<T> const & rhs);
octonion<T> & operator /= (::boost::math::quaternion<T> const & rhs);
template<typename X>
octonion<T> & operator /= (octonion<X> const & rhs);
```

These perform the mathematical operation `(*this)*inverse_of(rhs)` in this order (order is important as multiplication is not commutative for octonions) and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw. As for the multiplication, remember to group any two factors using parenthesis.

Octonion Non-Member Operators

Unary Plus and Minus Operators

```
template<typename T> octonion<T> operator + (octonion<T> const & o);
```

This unary operator simply returns o.

```
template<typename T> octonion<T> operator - (octonion<T> const & o);
```

This unary operator returns the opposite of o.

Binary Addition Operators

```
template<typename T> octonion<T> operator + (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator + (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator + (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator + (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> octonion<T> operator + (::boost::math::quaternion<T> const & lhs, octonion<T> const
template<typename T> octonion<T> operator + (octonion<T> const & lhs, ::boost::math::quaternion<T> const
template<typename T> octonion<T> operator + (octonion<T> const & lhs, octonion<T> const & rhs);
```

These operators return `octonion<T>(lhs) += rhs`.

Binary Subtraction Operators

```
template<typename T> octonion<T> operator - (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator - (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator - (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator - (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> octonion<T> operator - (::boost::math::quaternion<T> const & lhs, octonion<T> const
template<typename T> octonion<T> operator - (octonion<T> const & lhs, ::boost::math::quaternion<T> const
template<typename T> octonion<T> operator - (octonion<T> const & lhs, octonion<T> const & rhs);
```

These operators return `octonion<T>(lhs) -= rhs`.

Binary Multiplication Operators

```
template<typename T> octonion<T> operator * (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator * (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator * (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator * (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> octonion<T> operator * (::boost::math::quaternion<T> const & lhs, octonion<T> const
template<typename T> octonion<T> operator * (octonion<T> const & lhs, ::boost::math::quaternion<T> const
template<typename T> octonion<T> operator * (octonion<T> const & lhs, octonion<T> const & rhs);
```

These operators return `octonion<T>(lhs) *= rhs`.

Binary Division Operators

```
template<typename T> octonion<T> operator / (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator / (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator / (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator / (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> octonion<T> operator / (::boost::math::quaternion<T> const & lhs, octonion<T> const
template<typename T> octonion<T> operator / (octonion<T> const & lhs, ::boost::math::quaternion<T> const
template<typename T> octonion<T> operator / (octonion<T> const & lhs, octonion<T> const & rhs);
```

These operators return `octonion<T>(lhs) /= rhs`. It is of course still an error to divide by zero...

Binary Equality Operators

```
template<typename T> bool operator == (T const & lhs, octonion<T> const & rhs);
template<typename T> bool operator == (octonion<T> const & lhs, T const & rhs);
template<typename T> bool operator == (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> bool operator == (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator == (::boost::math::quaternion<T> const & lhs, octonion<T> const & rhs
template<typename T> bool operator == (octonion<T> const & lhs, ::boost::math::quaternion<T> const & rhs
template<typename T> bool operator == (octonion<T> const & lhs, octonion<T> const & rhs);
```

These return true if and only if the four components of `octonion<T>(lhs)` are equal to their counterparts in `octonion<T>(rhs)`. As with any floating-type entity, this is essentially meaningless.

Binary Inequality Operators

```
template<typename T> bool operator != (T const & lhs, octonion<T> const & rhs);
template<typename T> bool operator != (octonion<T> const & lhs, T const & rhs);
template<typename T> bool operator != (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> bool operator != (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator != (::boost::math::quaternion<T> const & lhs, octonion<T> const & rhs
template<typename T> bool operator != (octonion<T> const & lhs, ::boost::math::quaternion<T> const & rhs
template<typename T> bool operator != (octonion<T> const & lhs, octonion<T> const & rhs);
```

These return true if and only if `octonion<T>(lhs) == octonion<T>(rhs)` is false. As with any floating-type entity, this is essentially meaningless.

Stream Extractor

```
template<typename T, typename charT, class traits>
::std::basic_istream<charT,traits> & operator >> (::std::basic_istream<charT,traits> & is, octonion<T> &
```

Extracts an octonion `o`. We accept any format which seems reasonable. However, since this leads to a great many ambiguities, decisions were made to lift these. In case of doubt, stick to lists of reals.

The input values must be convertible to T. If bad input is encountered, calls `is.setstate(ios::failbit)` (which may throw `ios::failure` (27.4.5.3)).

Returns `is`.

Stream Inserter

```
template<typename T, typename charT, class traits>
::std::basic_ostream<charT,traits> & operator << (::std::basic_ostream<charT,traits> & os, octonion<T> c
```

Inserts the octonion `o` onto the stream `os` as if it were implemented as follows:

```
template<typename T, typename charT, class traits>
::std::basic_ostream<charT,traits> & operator << ( ::std::basic_ostream<charT,traits> & os,
octonion<T> const & o)
{
   ::std::basic_ostringstream<charT,traits> s;

   s.flags(os.flags());
   s.imbue(os.getloc());
   s.precision(os.precision());

   s << '(' << o.R_component_1() << ','
      << o.R_component_2() << ','
      << o.R_component_3() << ','
      << o.R_component_4() << ','
      << o.R_component_5() << ','
      << o.R_component_6() << ','
      << o.R_component_7() << ','
      << o.R_component_8() << ')';

   return os << s.str();
}
```

Octonion Value Operations

Real and Unreal

```
template<typename T> T  real(octonion<T> const & o);
template<typename T> octonion<T> unreal(octonion<T> const & o);
```

These return `o.real()` and `o.unreal()` respectively.

conj

```
template<typename T> octonion<T> conj(octonion<T> const & o);
```

This returns the conjugate of the octonion.

sup

```
template<typename T> T   sup(octonion<T> const & o);
```

This return the sup norm (the greatest among `abs(o.R_component_1())...abs(o.R_component_8())`) of the octonion.

l1

```
template<typename T> T   l1(octonion<T> const & o);
```

This return the l1 norm (`abs(o.R_component_1())+...+abs(o.R_component_8())`) of the octonion.

abs

```
template<typename T> T   abs(octonion<T> const & o);
```

This return the magnitude (Euclidian norm) of the octonion.

norm

```
template<typename T> T  norm(octonion<T>const  & o);
```

This return the (Cayley) norm of the octonion. The term "norm" might be confusing, as most people associate it with the Euclidian norm (and quadratic functionals). For this version of (the mathematical objects known as) octonions, the Euclidian norm (also known as magnitude) is the square root of the Cayley norm.

Quaternion Creation Functions

```
template<typename T> octonion<T> spherical(T const & rho, T const & theta, T const & phi1, T const & phi
template<typename T> octonion<T> multipolar(T const & rho1, T const & theta1, T const & rho2, T const &
template<typename T> octonion<T> cylindrical(T const & r, T const & angle, T const & h1, T const & h2, T
```

These build octonions in a way similar to the way polar builds complex numbers, as there is no strict equivalent to polar coordinates for octonions.

`spherical` is a simple transposition of `polar`, it takes as inputs a (positive) magnitude and a point on the hypersphere, given by three angles. The first of these, *theta* has a natural range of -pi to +pi, and the other two have natural ranges of -pi/2 to +pi/2 (as is the case with the usual spherical coordinates in $\boldsymbol{R}^3$). Due to the many symmetries and periodicities, nothing untoward happens if the magnitude is negative or the angles are outside their natural ranges. The expected degeneracies (a magnitude of zero ignores the angles settings...) do happen however.

`cylindrical` is likewise a simple transposition of the usual cylindrical coordinates in $\boldsymbol{R}^3$, which in turn is another derivative of planar polar coordinates. The first two inputs are the polar coordinates of the first $\boldsymbol{C}$ component of the octonion. The third and fourth inputs are placed into the third and fourth $\boldsymbol{R}$ components of the octonion, respectively.

`multipolar` is yet another simple generalization of polar coordinates. This time, both $\boldsymbol{C}$ components of the octonion are given in polar coordinates.

In this version of our implementation of octonions, there is no analogue of the complex value operation arg as the situation is somewhat more complicated.

Octonions Transcendentals

There is no `log` or `sqrt` provided for octonions in this implementation, and `pow` is likewise restricted to integral powers of the exponent. There are several reasons to this: on the one hand, the equivalent of analytic continuation for octonions ("branch cuts") remains to be investigated thoroughly (by me, at any rate...), and we wish to avoid the nonsense introduced in the standard by exponentiations of complexes by complexes (which is well defined, but not in the standard...). Talking of nonsense, saying that `pow(0,0)` is "implementation defined" is just plain brain-dead...

We do, however provide several transcendentals, chief among which is the exponential. That it allows for a "closed formula" is a result of the author (the existence and definition of the exponential, on the octonions among others, on the other hand, is a few centuries old). Basically, any converging power series with real coefficients which allows for a closed formula in $\boldsymbol{C}$ can be transposed to $\boldsymbol{O}$. More transcendentals of this type could be added in a further revision upon request. It should be noted that it is these functions which force the dependency upon the boost/math/special_functions/sinc.hpp and the boost/math/special_functions/sinhc.hpp headers.

exp

```
template<typename T>
octonion<T> exp(octonion<T> const & o);
```

Computes the exponential of the octonion.

cos

```
template<typename T>
octonion<T> cos(octonion<T> const & o);
```

Computes the cosine of the octonion

sin

```
template<typename T>
octonion<T> sin(octonion<T> const & o);
```

Computes the sine of the octonion.

tan

```
template<typename T>
octonion<T> tan(octonion<T> const & o);
```

Computes the tangent of the octonion.

cosh

```
template<typename T>
octonion<T> cosh(octonion<T> const & o);
```

Computes the hyperbolic cosine of the octonion.

sinh

```
template<typename T>
octonion<T> sinh(octonion<T> const & o);
```

Computes the hyperbolic sine of the octonion.

tanh

```
template<typename T>
octonion<T> tanh(octonion<T> const & o);
```

Computes the hyperbolic tangent of the octonion.

pow

```
template<typename T>
octonion<T>  pow(octonion<T> const & o, int n);
```

Computes the n-th power of the octonion q.

Test Program

The octonion_test.cpp test program tests octonions specialisations for float, double and long double (sample output).

If you define the symbol BOOST_OCTONION_TEST_VERBOSE, you will get additional output (verbose output); this will only be helpfull if you enable message output at the same time, of course (by uncommenting the relevant line in the test or by adding --log_level=messages to your command line,...). In that case, and if you are running interactively, you may in addition define the symbol BOOST_INTERACTIVE_TEST_INPUT_ITERATOR to interactively test the input operator with input of your choice from the standard input (instead of hard-coding it in the test).

Acknowledgements

The mathematical text has been typeset with Nisus Writer. Jens Maurer has helped with portability and standard adherence, and was the Review Manager for this library. More acknowledgements in the History section. Thank you to all who contributed to the discussion about this library.

History

- 1.5.8 - 17/12/2005: Converted documentation to Quickbook Format.
- 1.5.7 - 25/02/2003: transitionned to the unit test framework; <boost/config.hpp> now included by the library header (rather than the test files), via <boost/math/quaternion.hpp>.
- 1.5.6 - 15/10/2002: Gcc2.95.x and stlport on linux compatibility by Alkis Evlogimenos (alkis@routescience.com).
- 1.5.5 - 27/09/2002: Microsoft VCPP 7 compatibility, by Michael Stevens (michael@acfr.usyd.edu.au); requires the /Za compiler option.
- 1.5.4 - 19/09/2002: fixed problem with multiple inclusion (in different translation units); attempt at an improved compatibility with Microsoft compilers, by Michael Stevens (michael@acfr.usyd.edu.au) and Fredrik Blomqvist; other compatibility fixes.
- 1.5.3 - 01/02/2002: bugfix and Gcc 2.95.3 compatibility by Douglas Gregor (gregod@cs.rpi.edu).
- 1.5.2 - 07/07/2001: introduced namespace math.
- 1.5.1 - 07/06/2001: (end of Boost review) now includes <boost/math/special_functions/sinc.hpp> and <boost/math/special_functions/sinhc.hpp> instead of <boost/special_functions.hpp>; corrected bug in sin (Daryle Walker); removed check for self-assignment (Gary Powel); made converting functions explicit (Gary Powel); added overflow guards for division operators and abs (Peter Schmitteckert); added sup and l1; used Vesa Karvonen's CPP metaprograming technique to simplify code.
- 1.5.0 - 23/03/2001: boostification, inlining of all operators except input, output and pow, fixed exception safety of some members (template version).
- 1.4.0 - 09/01/2001: added tan and tanh.
- 1.3.1 - 08/01/2001: cosmetic fixes.
- 1.3.0 - 12/07/2000: pow now uses Maarten Hilferink's (mhilferink@tip.nl) algorithm.
- 1.2.0 - 25/05/2000: fixed the division operators and output; changed many signatures.

- 1.1.0 - 23/05/2000: changed sinc into sinc_pi; added sin, cos, sinh, cosh.
- 1.0.0 - 10/08/1999: first public version.

To Do

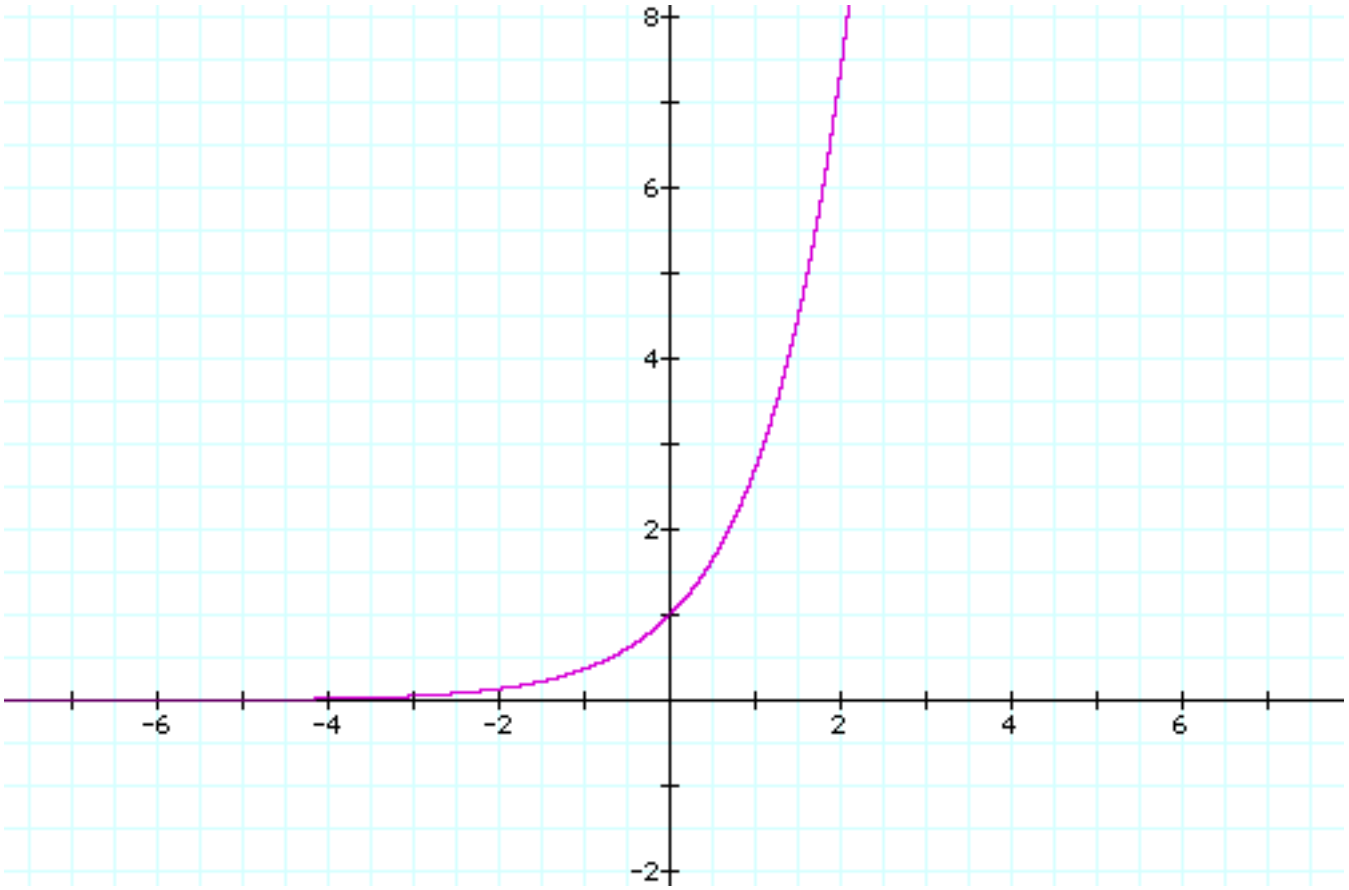
- Improve testing.
- Rewrite input operatore using Spirit (creates a dependency).
- Put in place an Expression Template mechanism (perhaps borrowing from uBlas).

Background Information and White Papers

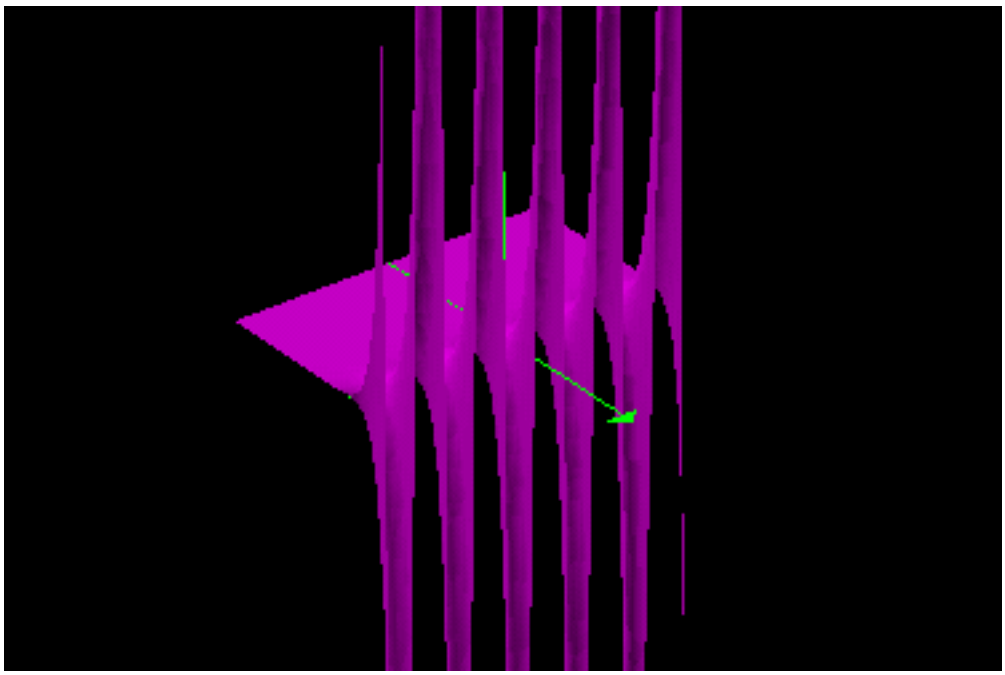
The Inverse Hyperbolic Functions

The exponential funtion is defined, for all object for which this makes sense, as the power series $\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!}$, with `n!` = `1x2x3x4x5...xn` (and `0!` = `1` by definition) being the factorial of `n`. In particular, the exponential function is well defined for real numbers, complex number, quaternions, octonions, and matrices of complex numbers, among others.

Graph of exp on R



Real and Imaginary parts of exp on C



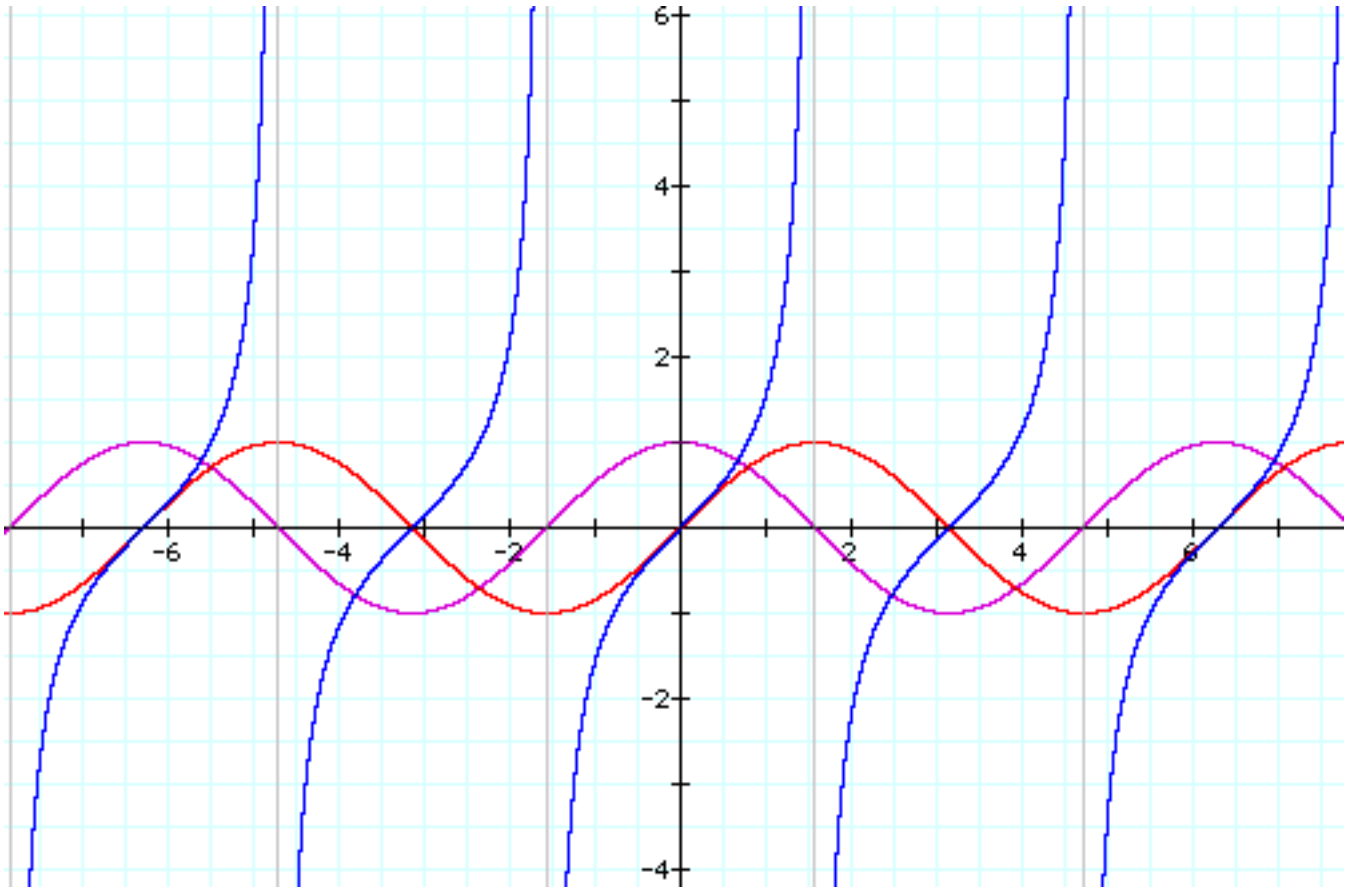
The hyperbolic functions are defined as power series which can be computed (for reals, complex, quaternions and octonions) as:

Hyperbolic cosine: $$\cosh(x) = \frac{\exp(+x) + \exp(-x)}{2}$$

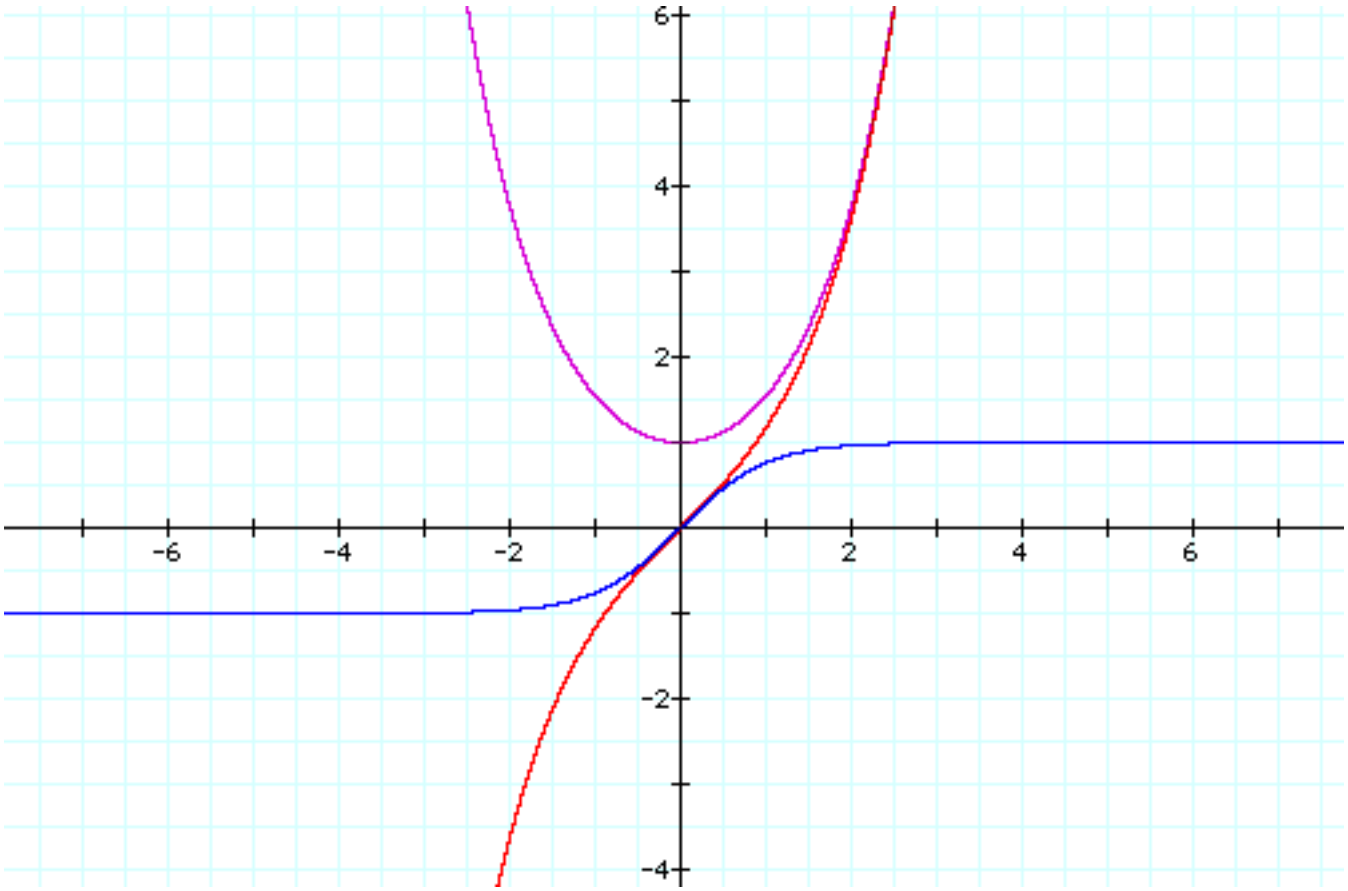
Hyperbolic sine: $$\sinh(x) = \frac{\exp(+x) - \exp(-x)}{2}$$

Hyperbolic tangent: $$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$$

Trigonometric functions on R (cos: purple; sin: red; tan: blue)



Hyperbolic functions on r (cosh: purple; sinh: red; tanh: blue)



The hyperbolic sine is one to one on the set of real numbers, with range the full set of reals, while the hyperbolic tangent is also one to one on the set of real numbers but with range `]-1;1[`, and therefore both have inverses. The hyperbolic cosine is one to one from `[0;+ [` onto `[+1;+ [` (and from `]- ;0]` onto `[+1;+ [`); the inverse function we use here is defined on `[+1;+ [` with range `[0;+ [`.

The inverse of the hyperbolic tangent is called the Argument hyperbolic tangent, and can be computed as $\operatorname{atanh}(x) = \frac{\log\left(\frac{1+x}{1-x}\right)}{2}$.

The inverse of the hyperbolic sine is called the Argument hyperbolic sine, and can be computed (for `x 0`) as $\operatorname{asinh}(x) = \log\left(x + \sqrt{x^2 + 1}\right)$.

The inverse of the hyperbolic cosine is called the Argument hyperbolic cosine, and can be computed as $\operatorname{asinh}(x) = \log\left(x + \sqrt{x^2 - 1}\right)$.

Sinus Cardinal and Hyperbolic Sinus Cardinal Functions

$$\operatorname{sinc}_a(x) = \frac{\sin\left(\frac{\pi x}{a}\right)}{\frac{\pi x}{a}}$$

The Sinus Cardinal family of functions (indexed by the family of indices `a > 0`) is defined by ; it sees heavy use in signal processing tasks.

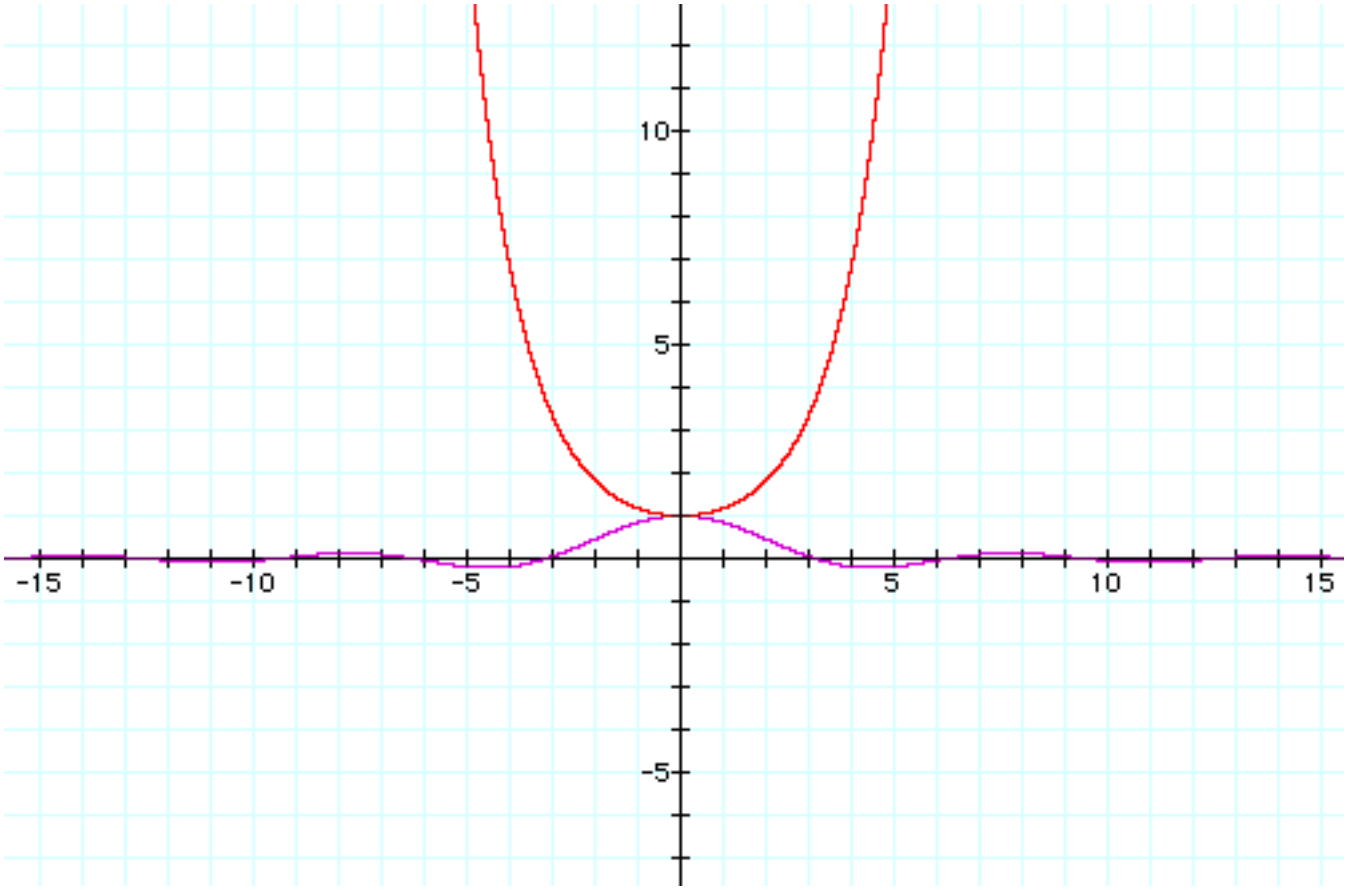
By analogy, the Hyperbolic Sinus Cardinal family of functions (also indexed by the family of indices `a > 0`) is defined by

$$\mathrm{sinhc}_a(x) = \frac{\sinh\left(\frac{\pi x}{a}\right)}{\frac{\pi x}{a}}$$

.

These two families of functions are composed of entire functions.

Sinus Cardinal of index pi (purple) and Hyperbolic Sinus Cardinal of index pi (red) on R



The Quaternionic Exponential

Please refer to the following PDF's:

- The Quaternionic Exponential (and beyond)
- The Quaternionic Exponential (and beyond) ERRATA & ADDENDA